

Анализ программ - Lab2

Матвей Русаков m.rusakov@innopolis.university SD-03

Апрель 2025

Предисловие

Материал и скриншоты из гидры я разместил на GitHub, в директории Lab2/lab_data/ вы можете найти исходник, скриншоты и сишный код мейн функции для каждой из задач

Task 0

Мейн функция нулевой задачи выглядит вот так:

```
undefined8 FUN_00101170(void)

{
    int iVar1;
    char *__s1;

    __s1 = (char *)malloc(400);
    printf("%s","Hello, enter the flag:\n");
    __isoc99_scanf(&DAT_00102004,__s1);
    iVar1 = strcmp(__s1,"flag{6057f13c496ecf7fd777ceb9e79ae2 85}");
    if (iVar1 == 0) {
        printf("%s",&DAT_00102046);
    }
    else {
        printf("%s","TRY HARDER");
    }
    return 0;
}
```

Описание

Данный код — это функция на языке C, реализующая проверку флага. Опишем её по шагам:

1. Выделяется память под строку:

```
__s1 = (char *)malloc(400);
```

Здесь выделяются 400 байт для хранения пользовательского ввода.

2. Печатается приглашение пользователю:

```
printf("%s", "Hello, enter the flag:\n");
```

На экран выводится сообщение:

```
Hello, enter the flag:
```

3. Считывается ввод:

```
__isoc99_scanf(&DAT_00102004, __s1);
```

4. Введённая строка сравнивается с жёстко закодированным флагом:

```
iVar1 = strcmp(__s1, "flag{6057f13c496ecf7fd777ceb9e79ae285}");
```

Если строки совпадают (`iVar1 == 0`), выполняется блок `if`, иначе — `else`.

5. Условие:

```
if (iVar1 == 0)
    printf("%s", &DAT_00102046);
else
    printf("%s", "TRY HARDER");
```

Если строка верная, выводится сообщение по адресу `&DAT_00102046` (строка "WIN"), иначе выводится сообщение "TRY HARDER".

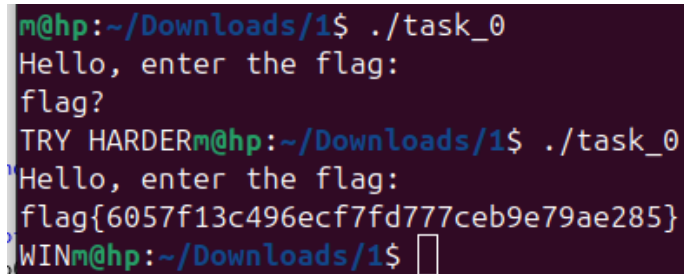
6. Функция возвращает 0:

```
return 0;
```

Особенности

- Отсутствует освобождение памяти после `malloc` (утечка памяти);
- Формат ввода не позволяет вводить пробелы (если используется `%s`);
- Сравнение производится напрямую, без шифрования или дополнительных преобразований.

Тестовые запуски



```
m@hp:~/Downloads/1$ ./task_0
Hello, enter the flag:
flag?
TRY HARDERm@hp:~/Downloads/1$ ./task_0
Hello, enter the flag:
flag{6057f13c496ecf7fd777ceb9e79ae285}
WINm@hp:~/Downloads/1$
```

Task 1

Как выглядит мейн функция - вы можете найти на гитхабе в референсах. Она слишком длинная, чтобы записывать ее в репорт

Описание Данный код — это функция на языке C, которая реализует проверку флага, аналогичную предыдущей, но с поэтапной посимвольной проверкой. Опишем её по шагам:

1. Объявление локальных переменных и инициализация:

```
int iVar1;
int local_40;
int local_3c;
char local_38 [4];
char cStack_34;
char cStack_33;
...
char acStack_13 [7];
undefined4 local_c;

local_c = 0;
local_3c = 0;
```

2. Приветственное сообщение:

```
printf("%s",
"Hello, this task is very similar to the
previous one, but has some
modifications\nenter the flag:\n"
);
```

Выводит сообщение с просьбой ввести флаг.

3. Чтение посимвольного ввода (38 символов):

```
for (local_40 = 0; local_40 < 0x26; local_40 = local_40 + 1) {
    __isoc99_scanf(&DAT_00102069, local_38 + local_40);
}
```

4. Пошаговое посимвольное сравнение (каждый символ сравнивается отдельно через `strcmp`), например:

```
iVar1 = strcmp("f", local_38, 1);
if (iVar1 == 0) {
    local_3c = 1;
    iVar1 = strcmp("l", local_38 + 1, 1);
    if (iVar1 == 0) {
        local_3c = 2;
        iVar1 = strcmp("a", local_38 + 2, 1);
        if (iVar1 == 0) {
            local_3c = 3;
            ...
        }
    }
}
```

Проверка продолжается символ за символом:

```
...
iVar1 = strcmp("}", acStack_13, 1);
if (iVar1 == 0) {
    local_3c = 0x26;
}
```

5. Финальная проверка:

```
if (local_3c == 0x26) {
    printf("%s", &DAT_00102097);
}
else {
    printf("%s", "TRY HARDER");
}
```

6. Возврат из функции:

```
return 0;
```

Отличия и особенности

- task_0 считывает строку - после enter или пробела запустится скрипт дальше, в то время как в task_1 используется посимвольный ввод
- Поскольку флаг состоит из 38 символов, функция будет ждать, пока пользователь не введет 38 символов и только потом запустит дальше
- В task_1 последовательное, посимвольное сравнение с буквами из правильного флага
- Флаг для этой задачи немного другой - flag{444Y0urB4seRBe803g2Usdfd4ds9y1re}

Тестовые запуски

```
m@hp:~/Downloads/1$ ./task_1
Hello, this task is very similar to the previous one, but has some modifications
enter the flag:
hello
this
is
my
flag
idk
ineed38symbols
to
exe
TRY HARDERm@hp:~/Downloads/1$ ./task_1
Hello, this task is very similar to the previous one, but has some modifications
enter the flag:
flag{444Y0urB4seRBe803g2Usdfd4ds9y1re}
WINm@hp:~/Downloads/1$
```

Task 2

Мейн В этой задаче исходный код функции отсутствует, программа не запускается, но есть информация, которую можно посмотреть в ghidra

Описание

1. Попытка запуска исполняемого файла без прав:

```
m@hp:~/Downloads/1$ ./task_2
bash: ./task_2: Permission denied
```

Вывод: нет прав на выполнение файла.

2. Попытка запуска через sudo:

```
m@hp:~/Downloads/1$ sudo ./task_2
[sudo] password for m:
sudo: ./task_2: command not found
```

Вывод: несмотря на `sudo`, команда не найдена. Файл не является исполняемым бинарником. Далее я посмотрел информацию о файле внутри ghidra, а также попробовал вывести все строки с помощью команды `strings`

3. Просмотр строк внутри файла через `strings`:

```
m@hp:~/Downloads/1$ strings task_2
Linux
Linux
1Hello!
1Bye-bye :(
license=flag{baee49fd4f7009ff6e932463791f28e6}
srcversion=FED0633F3F673540E886029
depends=
retpoline=Y
name=task_7
vermagic=6.2.0-34-generic SMP preempt mod_unload modversions
__fentry__
_printk
__x86_return_thunk
module_layout
task_7
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
.shstrtab
.note.gnu.build-id
.note.Linux
.text
.rodata.str1.1
__mcount_loc
.modinfo
.return_sites
.call_sites
__versions
__patchable_function_entries
.exit.data
.init.data
.gnu.linkonce.this_module
.bss
.comment
.note.GNU-stack
```

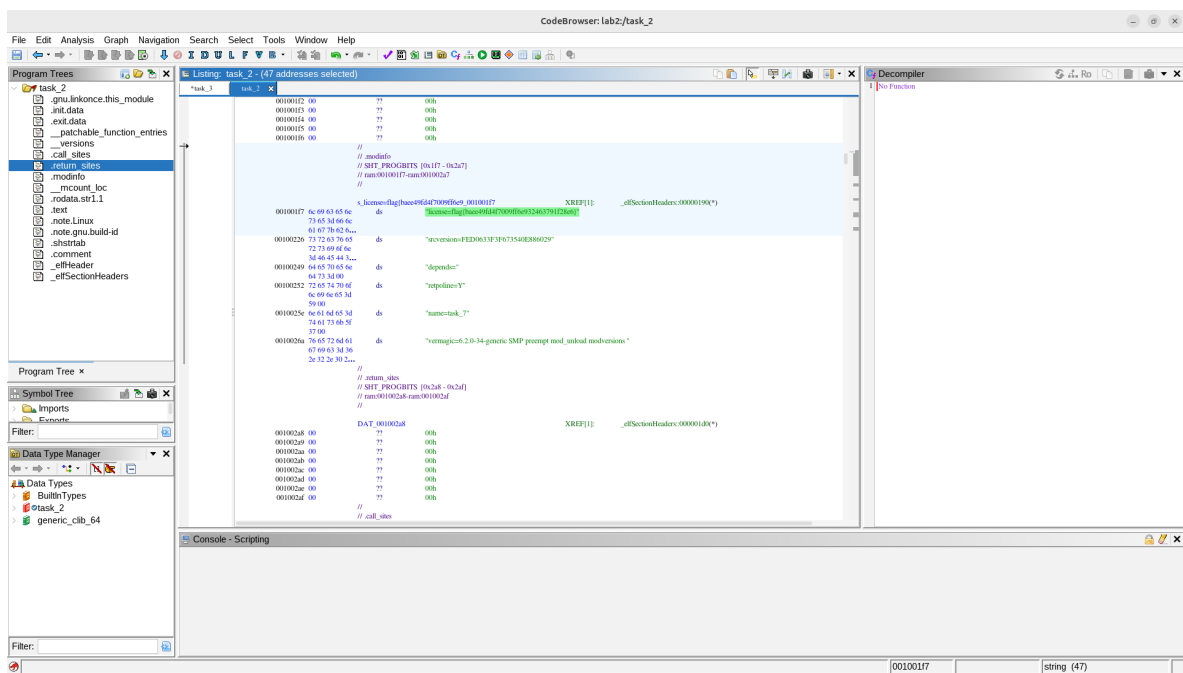
4. Ключевой момент — найденная строка:

```
license=flag{baee49fd4f7009ff6e932463791f28e6}
```

Особенности

- Файл `task_2` не является обычным исполняемым файлом, а представляет собой скомпилированный модуль ядра Linux.
- Запуск напрямую не работает, потому что модуль нужно загружать через `insmod` или `modprobe`, а не исполнять.
- Флаг хранится в строке с `license` внутри модуля.
- Флаг: `flag{baee49fd4f7009ff6e932463791f28e6}`
- В выводе присутствует `"depends= ...=task_7..."`, возможно этот файл содержит выводы для 7го задания типа "Hello!" и "Bye-bye :(" . Флаг, найденный здесь, возможно, подойдет для решения 7й задачи.

Тестовые запуски были в описании, поэтому прикреплю скриншот из гидры с полем лицензии и флага



Task 3

Мейн В этой задаче мы снова имеем дело с модулем ядра, который не запускается напрямую как исполняемый файл, но доступен для анализа через вывод команды `strings` и в ghidra.

Описание

1. Попытка запуска файла с правами суперпользователя:

```
m@hp:~/Downloads/1$ sudo ./task_3
[sudo] password for m:
sudo: ./task_3: command not found
```

Вывод: команда не найдена. Скорее всего, файл не является исполняемым бинарником, а модулем ядра.

2. Просмотр строк в файле через `strings`:

```
m@hp:~/Downloads/1$ strings task_3
Linux
Linux
AUATI
A]A^]1
flag
itmo
6[*] Bye - bye !
1reading...
1[*] Error assigning Major Number!
1[*] Failed to register device class
1[*] Failed to create the device
G'g/|
W[Osd
q-fn
rEdWcNDyavDSNOdKOC95iTEP8bioF3IPmAKUXx
license=GPL
description=find the flag
srcversion=0765CAD67B0F0DF80A62408
depends=
retpoline=Y
name=task_8
vermagic=6.2.0-34-generic SMP preempt mod_unload modversions
__register_chrdev
__class_create
device_create
__x86_return_thunk
_printk
class_destroy
```



```

__unregister_chrdev
device_destroy
class_unregister
vmalloc
__check_object_size
_copy_to_user
__copy_overflow
__fentry__
module_layout
task_8
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
__UNIQUE_ID_srcversion193
__UNIQUE_ID_depends192
____versions
__UNIQUE_ID_retpoline191
__UNIQUE_ID_name190
__UNIQUE_ID_vermagic189
_note_10
_note_9
intro_init
fops
__key.11
my_class
intro_exit
__UNIQUE_ID___addressable_cleanup_module241
__UNIQUE_ID___addressable_init_module240
__UNIQUE_ID_license239
__UNIQUE_ID_description238
__pfx_intro_read
__check_object_size
__class_create
__this_module
class_destroy
crypted
__fentry__
_printk
__copy_overflow
__pfx_init_module
major
device_create
class_unregister
__pfx_cleanup_module
__x86_return_thunk
_copy_to_user
__register_chrdev
device_destroy

```

```

vmalloc
__unregister_chrdev
.symtab
.strtab
.shstrtab
.note.gnu.build-id
.note.Linux
.rela.text
.rela.init.text
.rela.exit.text
.rodata.str1.1
.rodata.str1.8
.rela__mcount_loc
.rodata
.modinfo
.rela.return_sites
.rela.call_sites
__versions
.rela__patchable_function_entries
.rela.data
.rela.exit.data
.rela.init.data
.rela.gnu.linkonce.this_module
.bss
.comment
.note.GNU-stack

```

3. Анализ строк:

- В выводе присутствует строка **flag**, но не полный флаг.
- Присутствует строка, напоминающая зашифрованные данные:

```
rEdWcNDyavDSN0dKOC95iTEP8bioF3IPmAKUXx
```

- Также указано описание:

```
description=find the flag
```

Предполагаю, что флаг зашифрован и хранится в строке **crypted**.

4. После этого, в ghidra, через дизассемблированную функцию `__pfx_init_module` я нашел функцию `__pfx_intro_read`:

```

undefined1 [16] __pfx_intro_read(undefined8 param_1, undefined8 param_2, ulong param_3) {
long lVar1;
byte bVar2;

```

```

long lVar3;
byte bVar4;

_printk(&DAT_001006b3);
lVar1 = vmalloc(0x27);
bVar2 = 0x14;
bVar4 = 0x72;
lVar3 = 0;

while (true) {
    *(byte *)(lVar1 + lVar3) = bVar2 ^ bVar4;
    if (lVar3 + 1 == 0x26) break;
    bVar4 = "rEdWcNDyavDSN0dKOC95iTEP8bioF3IPmAKUXx"[lVar3 + 1];
    bVar2 = crypted[lVar3 + 1];
    lVar3 = lVar3 + 1;
}
*(undefined1 *)(lVar1 + 0x26) = 0;

if (param_3 < 0x28) {
    __check_object_size(lVar1, param_3, 1);
    __copy_to_user(param_2, lVar1, param_3);
} else {
    __copy_overflow(0x27, param_3);
}
return ZEXT816(0);
}

```

5. Размышления: Эта функция - дешифратор. Мы можем заметить тут наш зашифрованный флаг, который обнаружили ранее, а также массив `crypted`. Эта функция посимвольно делает XOR с элементами массива и расшифровывает флаг.
6. Массив `crypted` я нашел в сегменте `.rodata`. Теперь можно составить несложный питон скрипт для дешифровки флага:

```

crypted = [
    0x14, 0x29, 0x05, 0x30, 0x18, 0x7d, 0x70, 0x4a, 0x03, 0x47,
    0x27, 0x67, 0x2f, 0x7c, 0x01, 0x2a, 0x78, 0x71, 0x08, 0x57,
    0x5b, 0x30, 0x73, 0x64, 0x08, 0x04, 0x0a, 0x57, 0x71, 0x03,
    0x79, 0x34, 0x0f, 0x71, 0x2d, 0x66, 0x6e, 0x05
]

flag = "rEdWcNDyavDSN0dKOC95iTEP8bioF3IPmAKUXx"

decrypted = ''.join(chr(c ^ ord(f)) for c, f in zip(crypted, flag))

print(f"Decrypted flag: {decrypted}")

```

7. После дешифровки мы получаем наш флаг - flag{343b1c4a3ea721b2d640fc8700db0f36}

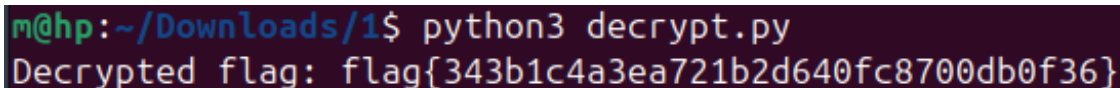
Особенности

- Файл `task_3` — это модуль ядра Linux.
- Флаг, зашифрован в строке:

```
rEdWcNDyavDSN0dKOC95iTEP8bioF3IPmAKUXx
```

- Метод шифрования - XOR
- После шифрования искомый флаг - flag{343b1c4a3ea721b2d640fc8700db0f36}
- В выводе присутствует "depends=task_8...", возможно этот файл содержит выводы для 8го задания

Тестовый запуск Оставлю тут вывод питоновского скрипта



```
m@hp:~/Downloads/1$ python3 decrypt.py
Decrypted flag: flag{343b1c4a3ea721b2d640fc8700db0f36}
```

Task 4

Мейн Бинарный файл (ELF, x86_64), в котором пользователь должен "угадывать" случайные числа.

```
undefined8 FUN_00101200(void)
{
    undefined *__s;
    uint in_EAX;
    time_t tVar1;
    long lVar2;
    size_t sVar3;
    int iVar4;
    int iVar5;
    int iVar6;
    ulong uVar7;
    undefined8 uStack_28;

    uStack_28 = (ulong)in_EAX;
```

```

puts("Hello, You have to predict random numbers:");
tVar1 = time((time_t *)0x0);
srandom((uint)tVar1);
iVar6 = 0x539;
iVar5 = 0;
do {
    __isoc99_scanf(&DAT_0010202b, (long)&uStack_28 + 4);
    lVar2 = random();
    __s = PTR_DAT_00104068;
    iVar4 = iVar5 + (uint)(lVar2 % 2 == (long)uStack_28._4_4_) * 2;
    iVar5 = iVar4 + -1;
    iVar6 = iVar6 + -1;
} while (iVar6 != 0);
if (iVar5 == 0x539) {
    if (*PTR_DAT_00104068 != '\0') {
        uVar7 = 0;
        do {
            putchar((int)(char)__s[uVar7] ^ iVar4 - 0x52dU);
            uVar7 = uVar7 + 1;
            sVar3 = strlen(__s);
        } while (uVar7 < sVar3);
    }
}
else {
    printf("Oh noo ...");
}
return 0;
}

```

Описание

- Бинарный файл выводит: "Hello, You have to predict random numbers:" Затем вызывает `srandom(time(NULL))`, делая генератор случайных чисел предсказуемым, если известен момент запуска.
- Цикл длин ой 1337 итераций, где:
- Генерируется случайное число `random()`.
- Пользователь вводит 0 или 1 (угадывает чётность).
- Если угадывает — увеличивается счётчик.
- После 1337 правильных угадываний сравнивается результат:
- Если все угадывания верны — расшифровывается строка `PTR_DAT_00104068` с помощью XOR.
- XOR ключ вычисляется как $(iVar4 - 0x52d) = 1338 - 1325 = 13$.

Используя команду `strings task_4` я получил что-то похожее на зашифрованный ключ: `kaljv44o<kk5k«:5<89<k:`

Я создал питоновский скрипт, который использует библиотеки `random` для симуляции рандомного поведения кода подбора букв и `time` для использования седа. Далее скрипт дешифровывает с XOR ключом искомый флаг. Найти скрипт можно по ссылке[1]

Особенности

- Задача построена на использовании `random()` из `glibc`, который инициализируется с помощью `srandom(time(NULL))`.
- Это предсказуемо, если у нас есть примерное время запуска программы.
- Зашифрованный ключ: `kaljv44o<kk5k«:5<89<k:k54k4oi9<n9l<:p`
- Дешифрованный ключ: `flag{99b1ff8f11781541f7f89f9bd41c4a17}`

Тестовый запуск Аутпут питоновского скрипта:

```
m@hp:~/PycharmProjects/reverse-engineering-course/Lab2/lab_data/task_4_data$ python3 decrypt.py
Found correct seed: 1746535932
Decrypted flag: flag{99b1ff8f11781541f7f89f9bd41c4a17}
```

Task 5

Мейн Данный ELF исполняемый файл не запускался на моём компьютере из-за ошибок с библиотекой `ryautogui` и отсутствия файла `task_4.py`. Для анализа использовался дизассемблер Ghidra.

Описание Программа `task_5` реализует загрузчик Python-приложения, упакованного с помощью `PyInstaller`. Основная задача - извлечь необходимые файлы во временный каталог, настроить окружение Python и запустить приложение.

Ключевые компоненты и функциональность

- Точка входа (`processEntry`)
 - Вызывает `__libc_start_main` для инициализации окружения
 - Передаёт управление основной функции `thunk_FUN_00403e50`
- Основная функция `thunk_FUN_00403e50`
 - Инициализация буферов и переменных
 - Проверка переменных окружения:
 - * `_MEIPASS2` - каталог с извлечёнными файлами
 - * `_PYI_ONEDIR_MODE` - режим работы `PyInstaller`
 - * `_PYI_PROcname` - имя процесса Linux (устанавливается через `prctl`)
- Функция инициализации `FUN_004086c0` - Конструктор, выполняющий `__DT_INIT_ARRAY` до запуска основной функции

- Основные функции загрузчика
 - Управление архивом PyInstaller
 - Управление переменными окружения
 - Обработка ошибок
 - Управление памятью и очистка временных ресурсов

Тестовый запуск

```
m@hp:~/Downloads/1$ ./task_5
Traceback (most recent call last):
  File "task_4.py", line 1, in <module>
ModuleNotFoundError: No module named 'pyautogui'
[31465] Failed to execute script 'task_4' due to unhandled exception!
m@hp:~/Downloads/1$
```

Task 6

Мейн информация Файл представляет собой ELF-бинарник под x86_64 Linux. При запуске программа предлагает пользователю выбрать один из двух режимов:

- **1. Encrypton** — зашифровать флаг из файла `flag.txt` и сохранить в `encrypted_flag.txt`
- **2. Decryption** — расшифровать содержимое `encrypted_flag.txt` и сохранить в `flag.txt`

Вход обрабатывается функцией `FUN_001011a0`, основной управляющей логикой занимается `FUN_00101460`.

Анализ шифрования Функция шифрования представлена в дизассемблированном виде в `FUN_00101280`. Она применяет к каждому байту входной строки позиционно-зависимое преобразование. Ниже приведена соответствующая формула:

$$\text{encrypted} = (i \oplus ((c - 0x19) \oplus \sim 0x28)) + 0x48$$

где:

- c — ASCII-код исходного символа
- i — индекс символа в строке

Пример кода на C (упрощённо):

```
char c = input[i];
char encrypted = (i ^ ((c - 0x19) ^ ~0x28)) + 0x48;
```

Элементы шифра:

- используется XOR с битовой маской
- учитывается позиция символа
- добавлены константные смещения (+0x19, +0x48 и др.)

Анализ дешифрования Дешифрование реализовано симметрично в FUN_00101380. Формула обратного преобразования выглядит так:

$$\text{original} = (((c - 0x29 - 0x1F) \oplus \sim 0x28) + 0x19) \oplus i$$

Эквивалентный код на C:

```
char c = input[i];
char decrypted = (((c - 0x48) ^ ~0x28) + 0x19) ^ i;
```

Зашифрованный флаг, прочитанный из файла, преобразуется обратно в оригинальную строку и сохраняется в flag.txt.

Особенности

- Используется самодельный криптоалгоритм с симметричным ключом.
- Преобразование для меня очень запутано, но линейно и обратимо.
- Для использования этого алгоритма обязательно использовать txt файлы для инпута и аутпута.

Тестовый запуск

```
m@hp:~/Downloads/1$ cat flag.txt
flag={mattwaynotencryptedflag}
m@hp:~/Downloads/1$ ./task_6
Hello, it's a custom crypto program
Select option:
1. Encrypton
2. Decryption
1
30f0l0a0g0={0m0a0t0t0w0a0y0n0o0t0e0n0c0r0y0p0t0e0d0f0l0a0g0}m@hp:~/Downloads/1$
./task_6
Hello, it's a custom crypto program
Select option:
1. Encrypton
2. Decryption
2
m@hp:~/Downloads/1$ cat flag.txt
0i h0m@hp:~/Downloads/1$ ./task_6
```

Предполагаю, что я что-то сделал не так, поскольку формулы шифрации и дешифрации противопоставляют друг другу

Task 7

Мейн информация Программа `task_7` является ELF исполняемым файлом. Это текстовый энкодер, принимающий строку текста в качестве аргумента.

Описание

- Если программа запускается корректно, выполнение продолжается.
- Если программа запускается некорректно, выводится сообщение: `"Usage: ./task_7 TEXT"`
- Программа выделяет память для локальных переменных.
- Для каждого символа входной строки выполняется последовательность операций:
 - К символу добавляется его позиция в строке (ADD)
 - Выполняется операция XOR с числом 14 (0xE)
 - Применяется маска через AND с числом 31 (0x1F)
 - Из результата вычитается (позиция + 1)
- Результат выводится в консоль в виде бинарных данных.

Тестовый запуск Здесь я попробовал зашифровать фразу "Hello" и расшифровать ее с помощью дешифратора из 6й задачи, но ничего не вышло(

```
m@hp:~/Downloads/1$ ./task_7 Hello | od -tx1 -v
00000000 05 06 fd fd 18
00000005
m@hp:~/Downloads/1$ ^C
m@hp:~/Downloads/1$ ./task_6
Hello, it's a custom crypto program
Select option:
1. Encrypton
2. Decryption
2
m@hp:~/Downloads/1$ cat flag.txt
XRm@hp:~/Downloads/1$
```

Также я так и не нашел зависимость `task_2` от `task_7`

Task 8

Мейн информация Программа `task_8` является исполняемым ELF-файлом. Для анализа была рассмотрена функция входа, поскольку она вызывает множество других функций, что затрудняет полное исследование всей программы. На основе анализа можно предположить, что `task_8` является частью системного приложения или программного обеспечения, связанного с обеспечением безопасности, учитывая наличие проверок, связанных с процессором.

2. Описание

2.1 Проверка процессора

Программа использует инструкцию `CPUID` для получения информации о процессоре. В частности, при вызове с параметром `leaf = 0` вызывается функция `cpuid_basic_info(0)`, результат которой сохраняется в указатель `piVar1`. Далее производится проверка идентификатора производителя процессора (`vendor ID`), сопоставляя полученные значения с ожидаемыми константами:

- `0x756e6547` ("Genu")
- `0x49656e69` ("ineI")
- `0x6c65746e` ("ntel")

Если идентификатор совпадает с "GenuineIntel", устанавливается флаг `DAT_0054fea9 = 1`, сигнализирующий о том, что процессор Intel.

2.2 Получение версии процессора

Далее программа вызывает `CPUID` с параметром `leaf = 1`, чтобы получить информацию о версии процессора через функцию `cpuid_Version_info(1)`. Результат сохраняется в переменную `DAT_0054ff04` для дальнейшего использования.

2.3 Логика инициализации

Процесс инициализации программы разделяется в зависимости от значения указателя `DAT_00520f48`:

- Если `DAT_00520f48 == NULL`, выполняется инициализация с использованием ранее полученной информации о процессоре. В частности, производится проверка значения `0x123`.
- Если `DAT_00520f48 != NULL`, происходит вызов функции через указатель с заданными параметрами, а также производится настройка смещений памяти.

2.4 Последовательные вызовы функций

В завершении выполняются последовательные вызовы функций `FUN_0045d1a0()` и `FUN_0045d160()`, которые делегируют выполнение функциям `FUN_00440360()` и `FUN_0043fe80()` соответственно. Эти функции включают:

- Реализацию цикла, проверяющего состояние стека относительно `FS_OFFSET`, для обеспечения целостности программы.

- Вызов функции `FUN_00458c20()` в случае нарушения условия целостности.
- Манипуляции с глобальными переменными и условные вызовы дополнительных функций для настройки состояния программы.
- Последовательность проверок, операций с блокировками (`LOCK/UNLOCK`), изменяющих значения контрольных переменных.
- Многоступенчатую цепочку условий, включающую арифметические, побитовые и числовые проверки, а также обработку специальных случаев с использованием операций `NaN` для сравнения чисел с плавающей точкой.
- При определённых условиях происходит вызов `FUN_00440160()`, `FUN_0045ab60()` и `FUN_0042fd40()` для завершения или корректировки работы программы.
- Функция `FUN_00440160()` реализует цикл проверки адреса стека с вызовом `FUN_00458c20()` при несоответствии, а также серию операций `LOCK/UNLOCK`, изменяющих глобальные переменные (например, `DAT_00550080`, `DAT_00550088`). Возвращаемое значение фиксировано (`0x2a`).
- Функция `FUN_0045ab60()` представляет собой пустую функцию-заглушку (`return;`), возможно, зарезервированную для будущей логики или проверки наличия вызова.
- Функция `FUN_0042fd40()` выполняет вызовы инициализации через `FUN_00458ae0()` и `FUN_0042ffa0()`, проверяет и устанавливает значение в структуре по смещению `0xf4`, а затем записывает значение `0` по адресу `uRam0000000000000000`.

3. Особенности

- Используется защита стека (через `in_FS_OFFSET`).
- Программа содержит структурированную обработку ошибок.
- Реализованы множественные проверки инициализации, обеспечивающие корректность запуска.
- Наблюдается сложная логика работы с глобальными переменными и флагами, а также использование атомарных операций для синхронизации.
- Программа реализует внутренние механизмы самопроверки и восстановления состояния при возникновении ошибок.
- Присутствуют функции-заглушки, не содержащие функциональной нагрузки, возможно, используемые как точки входа или метки для дальнейшего развития.
- Связь с `task_3` я также не нашел

References

- [1] GitHub Link: <https://github.com/MattWay224/reverse-engineering-course> В этом репозитории можно найти все лабы и информацию про каждое задание в каждой лабе