**Reverse 10 класс — 2**

**Описание**

Дан бинарь command-block , который ведёт себя по-разному в обычном режиме и под дебагером (anti-debug).

**Структура:**

В обычном режиме программа выводит **ложный хеш** (необратимое преобразование)

При запуске под дебагером выполняется код с реальным обратимым преобразованием

XOR ключ генерируется динамически из 10 частей в функции generate\_xor\_key чтобы было легче посмотреть его в дебагере, а не вычислять самому

**Задание:**

1. **ФЛАГ 1:** Найти XOR ключ в памяти программы ( vsosh{k3y\_g3n3r8d} )

2. **ФЛАГ 2:** Расшифровать файл output.txt с помощью найденного ключа(vsosh{g3t\_0u774\_h3r3\_u\_cr33p3r!!})

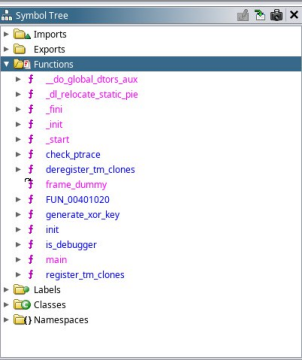
**Решение**

**Ghidra + python**

Открыв файл в ghidra и изучив main видно что от вводимого пользователя текста берется хэш, это значит что его перебор явно нецелесообразен.

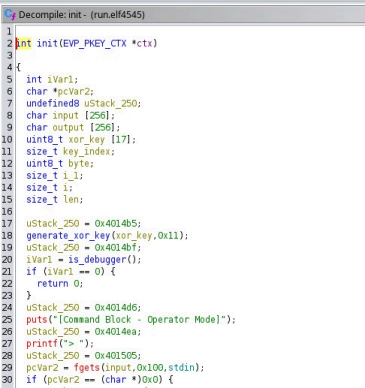
Погуляя по функциям можно найти функцию init , в которой находится настоящая логика преобразования, так же видно интересную функцию generate\_xor\_key, выполнением которого является ксор 10 захардкоженных массивов, запомним это,

очевидно, что это первый флаг:

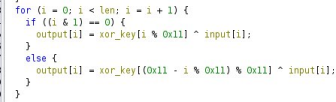


Чтобы понять где эта функция вызывается достаточно нажать на функцию правой кнопкой мыши, нажать на Show Reference to и увидеть там, что эта функция находится в \_\_init\_array\_entry , что указывает на то, что функция вызывается в конструкторе перед выполнением main .

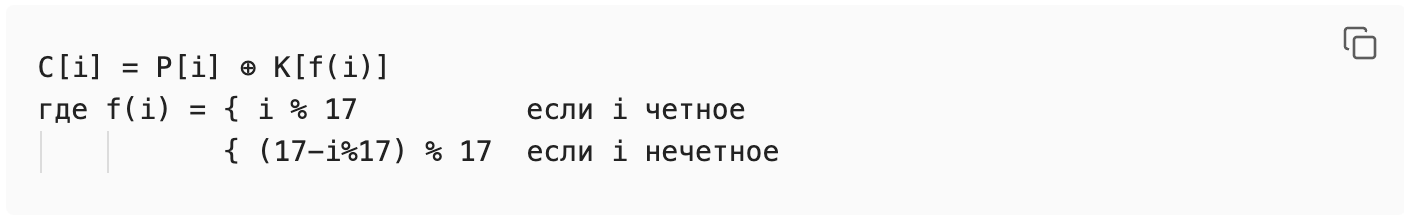
Так же наличие вызова функции is\_debugger (и check\_ptrace внутри) должно наталкивать на мысль, что спровоцировать выполнение основной логики можно только под дебагером:



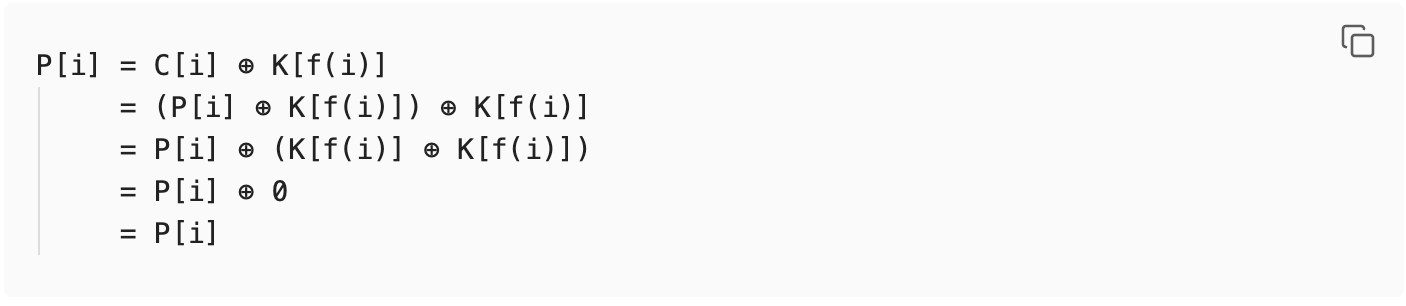
Теперь, ознакомимся с алгоритмом, по которому преобразуется пользовательский ввод:



Видим, что за преобразованием стоит ксор с элементами ключа с учетом четности индекса. Зная свойство обратимости ксора:



то, обращение будет выглядеть как:



Код:



Теперь, когда алгоритм преобразования ясен рассмотрим два способа получить xor\_key при помощи GDB и EDB

**GDB**

Самый простой способ. Нам нужно поставить breakpoint на функцию init и перешагнуть выполнение generate\_xor\_key , а дальше вывести его со стека

gdb run.elf

(gdb) break generate\_xor\_key

(gdb) run

Reading symbols from run.elf...

(gdb) break init

Breakpoint 1 at 0x4014a4: file run.c, line 93.

(gdb) run

Breakpoint 1, init () at run.c:93

93 generate\_xor\_key(xor\_key, sizeof(xor\_key));

(gdb) next

96 if (is\_debugger()) {

(gdb) print xor\_key

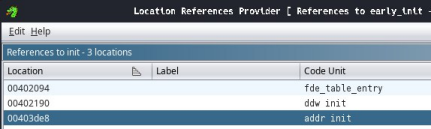
$1 = "vsosh{k3y\_g3n3r8d}"

**EDB**

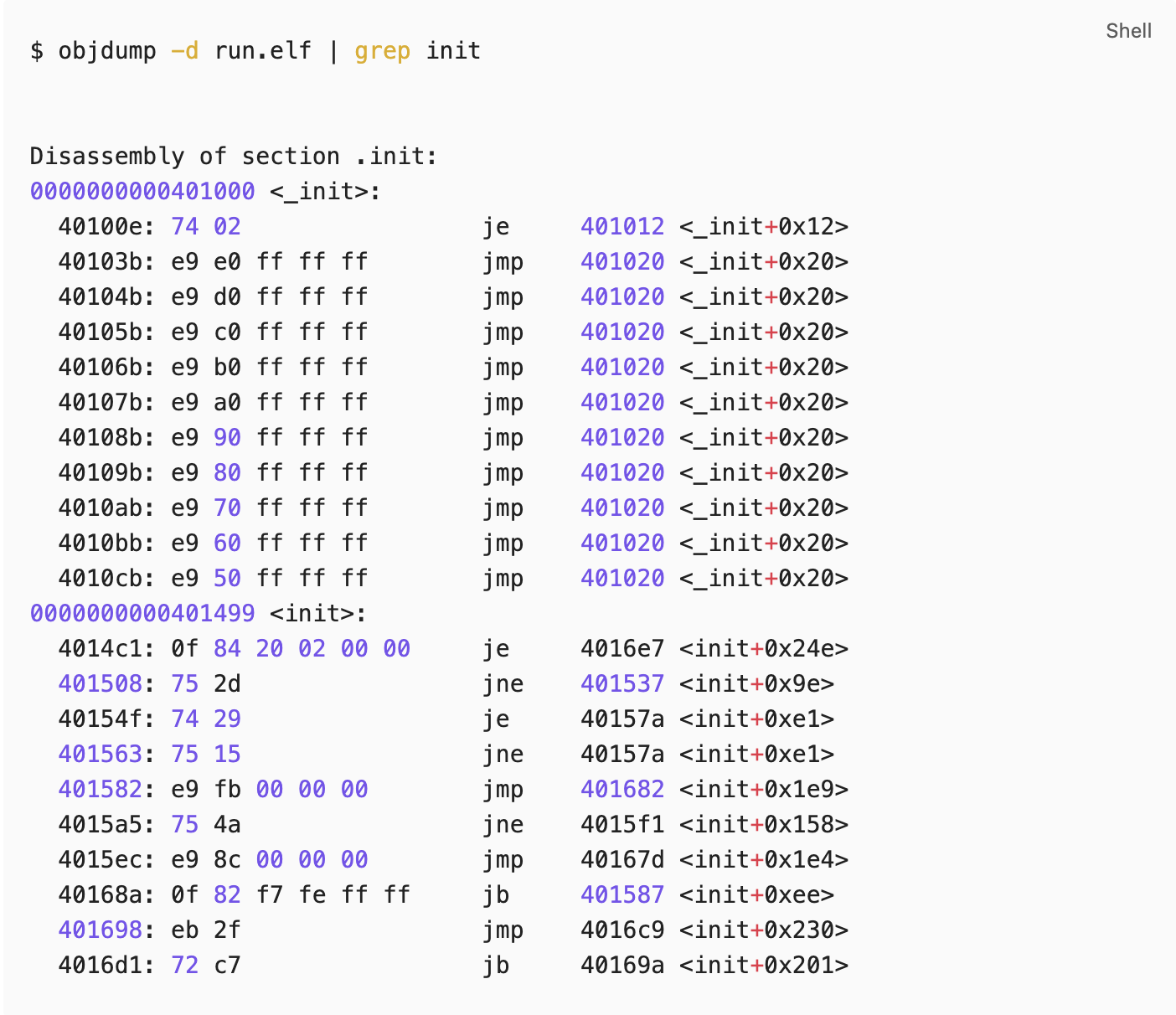
Чтобы поставить breakpoint сначала найдем адрес функции init , его можно посмотреть либо в Ghidra :

1. Нажимаем правой кнопкой мыши на функцию слева в списке Symbol Tree

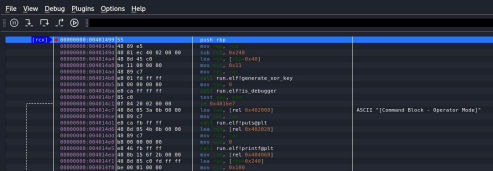
2. Show References to

3.  4.  5. Адрес хранится как little-endian, так что сам адрес будет 0x401499

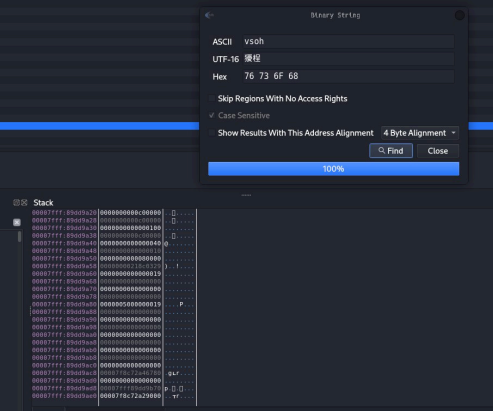
Легче можно посмотреть через objdump :



Теперь поставим breakpoint в edb. Для начала запустим edb --run ./command block.elf

Чтобы поставить breakpoint нужно открыть breakpointManager , его можно открыть либо в меню: Plugins/BreakpointManager , либо просто нажать Ctrl+B . Дальше Add Breakpoint , вводим адрес функции init : 0x401499 , закрываем BreakpointManager . Дальше нажимаем F9 чтобы программа выполнилась до init , либо нажимаем на кнопку выполнения сверху слева. Выглядит это следующим образом: Дальше нажимаем F8 (либо Step Out сверху слева) до тех пор пока не перешагнем

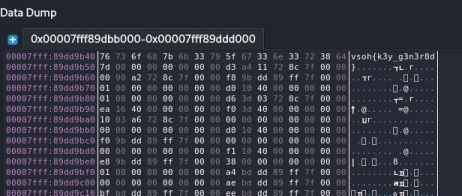
call run.elf!generate\_xor\_key , после чего тыкаем на область Stack снизу чтобы после этого нажать Ctrl+F и ввести в поиск строку vsosh



В результате поиска будет выдан адрес на стеке:



Кликнув по которому или перейдя( Ctrl+G ) увидим:

Теперь у нас есть все необходимое чтобы расшифровать output.txt.