**Reverse 11 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 :( и завершает выполнение При запуске под дебагером активируется **Operator Mode** с реальным обратимым преобразованием

Первый флаг генерируется динамически из 10 частей в функции summon\_creeper чтобы было легче посмотреть его в дебагере, а не вычислять самому

**Задание:**

1. **ФЛАГ 1:** Найти XOR ключ в памяти программы

( vsosh{n0\_w4y\_s0me0ne\_l3ft\_1t\_h3r3...} )

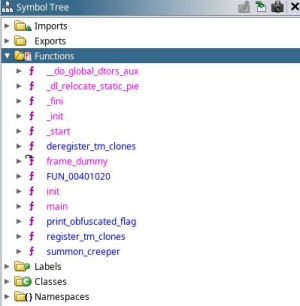
2. **ФЛАГ 2:** Заставить программу пройти все проверки и выполнить все проверки vsosh{w0w\_u\_4r3\_4dv4nc3d\_b1n4ry\_m1n3cr4ft3r!!!}

**Решение**

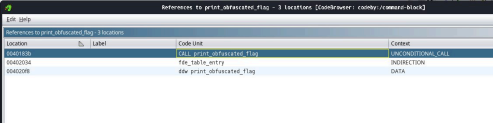
**Ghidra**

Запустив программу увидим что она просто выводит :( , откроем ghidra и посмотрим на main функцию:



В ней ничего не происходит, посмотрим на другие функции: 

Видим функцию, котороая однозначно намекает на свой функционал: print\_obfuscated\_flag , посмотрим на нее в дизассемблере:

видно, что функция как-то проверяет retaddr и передаваемый функции параметр чтобы вывести флаг, посмотрим откуда эта функция вызывается, в таблице выше нажмем на функцию правой кнопкой мыши и выберем Show References to : Перейдя по этому вызову окажемся в функции init :

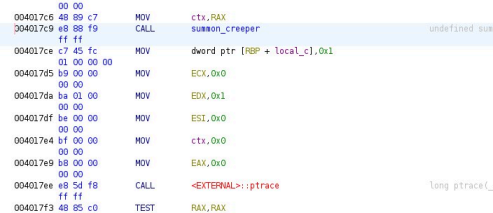


в которой видим две важные вещи:

1. ptrace

2. summon\_creeper

Провалившись в summon\_creeper увидим что она возвращает ксор десяти частей, этот результат потом передается в print\_obfuscated\_flag , проверка параметра ограничивается равенством первого байта на v , что значит что передаваемый параметр - первый флаг, посмотрим его в дебаггере: найдем где вызывается функция и поставим breakpoint после ее выполнения:

Поставим breakpoint на 0x4017ce : внутри функции увидим что результат лежит на

стеке, так что смотреть будем на нем:

(gdb) break \*0x4017ce

Breakpoint 1 at 0x4017ce

(gdb) run

Breakpoint 1, 0x00000000004017ce in init ()

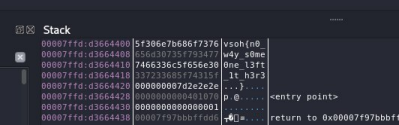
(gdb) x/s $rbp-0x30

0x7fffffffd630: "vsosh{n0\_w4y\_s0me0ne\_l3ft\_1t\_h3r3...}"

Так же можно посмотреть флаг на стеке через edb :

Поставить breakpoint( CTRL+b -> add breakpoint -> F9 )

Посмотреть на стек:

Нашли первый флаг, двигаемся дальше.

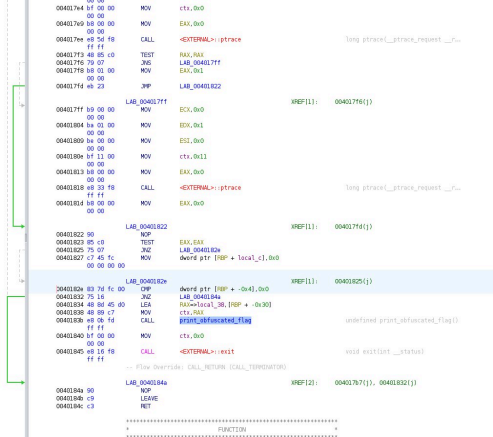
Теперь осознаем что происходит с ptrace , ptrace - это известный системный вызов linux для отладки и мониторинга процессов, когда мы вызываем ptrace c TRACE\_ME - он возвращает ошибку, т е -1 , если программа уже находится под отладкой(запущена под gdb , edb , ...). Дальше нам интересна проверка uVar2 = (uint)(-1 >= lVar3) , эта проверка должна быть true , т е 1 чтобы мы продвинулись дальше, это значит что это можно сделать под дебагером. Запомнили, смотрим дальше. Дальше мы видим абсолютно такую же проверку на дебаг как и раньше, но чтобы попасть в нее нам нужно чтобы ptrace(PTRACE\_TRACEME..) вернуло 0 , что значит означает что это невозможно находясь под дебагером, значит нужно

думать как можно сломать эту логику чтобы все-таки спровоцировать вызов print\_obfuscated\_flag .

Рассмотрим два варианта выхода из сложившейся ситуации:

**Debug**

Рассмотрим как эта функция выглядит в ассемблере:

Увидим что проверка, которую нам нужно пройти находится по адресу 0x401823 , нам

нужно не провалиться в jnz в следующией инструкции, значит нужно поставить breakpoint на этот адрес и сетнуть eax в ноль, и запустить исполнение дальше.

Cделаем это в gdb :

(gdb) break \*0x401823

Breakpoint 1 at 0x401823

(gdb) run

Breakpoint 1, 0x0000000000401823 in init ()

(gdb) set $eax = 0

(gdb) finish

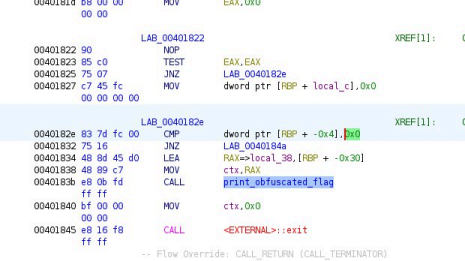
Run till exit from #0 0x0000000000401823 in init ()

vsosh{w0w\_u\_4r3\_4dv4nc3d\_b1n4ry\_m1n3cr4ft3r!!!}

**Patching**

Можно так же запатчить бинарь в ghidra:

Для этого нам нужно найти байт, в котором проверяется истинность значения и просто поменять его с нуля на единицу:



Нажать правой кнопкой мыши на 0x0

Patch instruction

0x0 -> 0x1

Сохранить бинарь

Дальше запускаем сохраненный бинарь под gdb:

(gdb) run

vsosh{w0w\_u\_4r3\_4dv4nc3d\_b1n4ry\_m1n3cr4ft3r!!!}

Цель работы: Использовать дебагер для манипуляции регистрами процессора и обхода второй проверки

Итог работы: Получить два флага:

**ФЛАГ 1**: Найти XOR ключ в памяти программы (можно извлечь в дебагере) **ФЛАГ 2**: Получить финальный флаг, изменив результат второй проверки

Критерий оценки: Предоставление обоих флагов и описание метода решения