Итак было дано 2 файла - исполняемый файл виртуальной машины и байткод от неё. Открываем первый файл в IDA Pro, нажимаем “tab”, чтобы открыть вывод декомпилятора и затем нажимаем “\”, чтобы скрыть преобразования типов.

```
if ( argc != 2 )
  error("Usage:\n\t./program_name [bytecode filename]", argv);
stream = fopen(argv[1], "r");
if ( !stream )
  error("Can't open file!!!", "r");
fseek(stream, 0LL, 2);
size = ftell(stream);
if ( size > 0x10000 )
  error("File is too big!", 0LL);
v3 = alloca(65552LL);
bytecode = (16 * ((&v6 + 3) >> 4));
fseek(stream, 0LL, 0);
fread(bytecode, size, 1uLL, stream);
```

В начале программа открывает файл переданный первым первым аргументом, проверяет его размер и читает содержимое в буфер, расположенный на стеке. Указатель на этот буфер располагается в глобальной переменной “bytecode”.

```
33  word_202134 = 0;
34  while ( 1 )
35  {
36    jump_performed = 0;
37    v8 = word_202134 + 1;
38    v7 = *(bytecode + word_202134);
39    v10 = v7 >> 4;
40    if ( v10 > 11 )
41    {
42      v13 = get_arg(1LL, &v8, v7);
43      (*(&op_funcs + v10))(v13);
44    }
45    else
46    {
47      for ( i = 0; i <= 1; ++i )
48      {
49        v4 = get_arg(i, &v8, v7);
50        *(&v14 + i) = v4;
51      }
52      if ( v7 & 1 )
53        v5 = &op_funcs16;
54      else
55        v5 = &op_funcs;
56      (v5[v10])(v14, v15);
57    }
58    if ( !jump_performed )
59      word_202134 = v8;
60  }
```

Затем идёт цикл, в котором из байткода берётся один байт , сохраняется в v7 и проверяются его первые 4 бита (“v10”). Если они больше 11 ти , то программа вызывает функцию “get_arg” один раз , после чего вызывает функцию, указатель на которую находиться в массиве “op_funcs” по индексу “v10” и передаёт ей значение, которое вернула функция “get_arg”. Получается, что v10 (первые 4 бита) - это номер инструкции для виртуальной машины. В случае если номер инструкции меньше или равен 11 ти , то программа вызывает функцию “get_arg” дважды и затем проверяет младший бит , в случае если он равен единице , то вызывается функция по адресу из массива “op_funcs16” , иначе вызывается из массива “op_funcs” и в любом из этих случаев функции передаются 2 значения, которые вернули вызовы “get_arg”. И в конце цикла проверяется, значение глобальной переменной “jump_performed”, в случае если она равна нулю, то переменной “word_202134” присваивается значение переменной v8.

Исходя из всего вышеперечисленного можно предположить, что:

1. Переменная word_202134 содержит смещение текущей инструкции относительно буфера с байткодом.
2. Переменная v8 под конец цикла содержит смещение следующей инструкции относительно того же буфера и изменяется в функции get_arg.
3. Старшие 4 бита содержат номер инструкции, а младший бит содержит тип инструкции (16 или 8 битная).

Давайте заглянем в код функции “get_arg”.

```
__int16 *__fastcall get_arg(int a1, _WORD *a2, char a3)
{
  char *v3; // rcx
  __int16 v4; // ax
  char *v5; // rcx
  __int16 v6; // ax
  __int16 *v8; // [rsp+18h] [rbp-8h]

  if ( (((a3 >> 1) & 7) >> a1) & 1 )
  {
    v3 = bytecode;
    v4 = (*a2)++;
    v8 = &regs[v3[v4]];
  }
  else if ( a1 == 1 && (a3 >> 1) & 4 )
  {
    if ( a3 & 1 )
    {
      imm = *(bytecode + *a2);
      *a2 += 2;
    }
    else
    {
      v5 = bytecode;
      v6 = (*a2)++;
      imm = v5[v6];
    }
    v8 = &imm;
  }
  else
  {
    v8 = (bytecode + *(bytecode + *a2));
    *a2 += 2;
  }
  return v8;
}
```

При вызове этой функции первым аргументом передаётся номер аргумента для инструкции , в случае , если номер инструкции больше 11 ти , то это всегда 1, иначе это 0 для первого и 1 для второго аргументов. Второй аргумент - это указатель на переменную в которой хранится смещение инструкции относительно буфера с байткодом. (Т.е указатель на v8 из функции main). И третий аргумент это байт из буфера bytecode по индексу “word_202134”. В первом условии проверяется 2+a1 младший бит (т.е 2 младший для первого аргумента и 3 ий для второго). В случае если он равен единице , то из байткода берётся один байт и возвращается указатель на элемент в массиве регистров по индексу этого байта. Иначе если аргумент - второй и 4 ый младший бит является единицей, то аргумент - это константное значение, его размер зависит от типа инструкции. Константное значение записывается в глобальную переменную и функция возвращает указатель на неё. И в случае если оба условия оказались ложными, то аргумент - это адрес и тогда возвращается адрес байткода +

два байта в little endian , находящиеся по смещению “*a2”. И теперь у нас есть полная информация о формате байткода.

1. Старшие 4 бита - номер инструкции.
2. После них идёт бит, который указывает является ли второй аргумент константным значение.
3. Затем располагаются 2 бита, которые указывают является ли аргумент регистром. (6 ой бит - первый аргумент , 7 ой бит - второй)
4. И в конце бит, который указывает является ли инструкция 16 битной.

Теперь имея формат необходимо написать декомпилятор для байткод и разобраться в нём. Код декомпилятора - https://pastebin.com/9jwShVbV. Запускаем его и видим код.

```
mrf@ubuntu:~/vm$ python3 decompiler.py b1 | head -n 25
0x4: MOV REG 1, 4352 (0x1100)
0x8: MOV REG 0, 4096 (0x1000)
0xa: INT 0 (0x0)
0xc: INT 1 (0x1)
0xf: MOV REG 2, 0 (0x0)
0x12: CMP REG 2, 17 (0x11)
0x14: JIT 50 (0x32)
0x18: LDB [0x100], REG 0
0x1c: LDB [0x101], REG 1
0x21: XOR [0x100], [0x101]
0x25: CMP [0x100], 255 (0xff)
0x27: JIF 58 (0x3a)
0x2a: ADD REG 0, 1 (0x1)
0x2d: ADD REG 1, 1 (0x1)
0x30: ADD REG 2, 1 (0x1)
0x32: JMP 15 (0xf)
0x36: MOV REG 0, 4864 (0x1300)
0x38: INT 0 (0x0)
0x3a: INT 2 (0x2)
0x3e: MOV REG 0, 4608 (0x1200)
0x40: INT 0 (0x0)
0x42: INT 2 (0x2)
0x47: MOV [0x0], [0x0]
0x4c: MOV [0x0], [0x0]
0x51: MOV [0x0], [0x0]
```

Этот код выводит строку “Enter password:‘, после чего читает её и в цикле выполняет операцию “исключающее или” между байтами прочитанной строки и байтами , находящимися по адресу 0x1100. Проводим обратные операции и получаем флаг.

CTF{VMs_4r3_c00l}