

# The Grand Reverse Engineering Challenge

В этом году с 21 мая проводилось соревнование **The Grand Reverse Engineering Challenge**, состоявшее из **2-х раундов**. В 1-ом раунде соревнования для решения предложены **7 задач**. Часть задач на тему **деобфускации** для своего решения требуют восстановление исходного кода ([Pic.1](#)).

Поставим своей целью - создание общей последовательности действий, позволяющей для каждого задания восстановить исходный код, выполнить его анализ и найти решение. (The goal is to deobfuscate the program and return it to idiomatic C)

Приступим к решению 1-ого задания. По условию требуется ввести два числа, которые вызовут аварийный останов программы. (A successful submission consists of two 32-bit numbers that, when read from standard input, cause the program to crash with a segmentation fault)

## Часть 1. Восстановление исходного кода.

### Шаг 1. Предварительный анализ исполняемого файла в декомпиляторе.

Требуется найти обfuscatedированный код, для которого характерным является большой и запутанный вид исполняемого графа. Выполним загрузку и автоматический анализ загруженного исполняемого файла задания в декомпиляторе **Idapro** (**Ghidra**).

Найдем функцию **main()**. Ее исполняемый график **маленький** ([Pic.2](#)). Вызываются три процедуры: **sub\_D55E()**, **sub\_1BF1()**, **sub\_21C5()** и несколько библиотечных функций, которые выполняют выделение памяти, ввод пользователем с клавиатуры 2-х чисел и вывод результат.

Из всех вызываемых процедур только **sub\_21C5()** обfuscatedирована, так как ее исполняемый график **большой и запутанный** ([Pic.3](#)).

### Шаг 2. Реализация обfuscatedированного кода.

Требуется создать код на языке си, реализующий обfuscatedированную процедуру.

Выполним генерацию **псевдокода** процедуры **sub\_21C5()**. Для анализа структуры и переменных выполним его генерацию в **Idapro**, а для обработки скриптами в **Ghidra**.

# Grand Reverse Engineering Challenge

The Grand Reverse Engineering Challenge runs in two rounds: **Round 1** starts **May 21** and **Round 2** starts **July 3**. The competition ends **midnight, anywhere on earth, July 18**. The total prize sum is USD 10,000. All individuals are welcome to participate. The only requirement is that you run our data collection framework in the background (typically within a Linux virtual machine) as you solve the challenges. The data collected will be published (anonymized, of course) and used to learn how reverse engineers solve problems in practice.

## Round 1

- **Challenge 1 ( LIGHT EXTRACT X86 ARM )**: challenge-1-x86\_64-Linux.exe and challenge-1-armv7-Linux.exe read two unsigned 32-bit numbers in decimal form from standard input and print a number to standard output. A successful submission consists of two 32-bit numbers that, when read from standard input, cause the program to crash with a segmentation fault. The original program is about 600 lines of C, the obfuscated program about 21,000 lines.
- **Challenge 2 ( MEDIUM DEOBFUSCATE X86 )**: challenge-2-x86\_64-Linux.exe and challenge-2-armv7-Linux.exe read two unsigned 32-bit numbers in decimal form from the command line and print a number to standard output. The goal is to deobfuscate the program and return it to idiomatic C. The original function is small, less than 100 lines of C.
- **Challenge 3 ( HEAVY DEOBFUSCATE X86 )**: challenge-3-x86\_64-Linux.exe takes 3 command line arguments, 32-bit unsigned integers in decimal form. For example, "challenge\_3\_linux\_x86\_64.exe 35:234792379 1100292587" will produce an output of the form "Answer for x = 35329, y = 234792379, z = 1100292587 is 3546740429". The code defines a piecewise mathematical function on the inputs. Your job is to figure out what that function is. The answer is not elegant, but it is short: each part of the function can be written in one line.
- **Challenge 4 ( HEAVY DEOBFUSCATE X86 )**: Challenge 4 ( challenge-4-x86\_64-Linux.exe ) is identical to Challenge 3, but uses a different type of protection.
- **Challenge 5 ( LIGHT EXTRACT ARM )**: challenge-5-armv7-Linux.exe takes a license key as its first argument on the command line. It is your job to find this key.
- **Challenge 6 ( MEDIUM EXTRACT ARM )**: challenge-6-armv7-Linux.exe is a protected version of the zip utility. A list of command-line arguments can be generated with "./challenge-6-armv7-Linux.exe --help". An additional, undocumented command-line argument (-K <some string>) can be used to pass a key to the program. The program only produces valid zip-files when this key is correct. It is your job to find the correct key.
- **Challenge 7 ( LIGHT TAMPER ARM X86 )**: challenge-7-x86\_64-Linux.exe and challenge-7-armv7-Linux.exe read two English words from standard input and print a number to standard output. The words are no longer than 12 characters, consist of the lower case letters a-z, and have been taken from this list. For two particular words the program will crash with a segmentation fault. Your task is to modify the program to remove the code that causes this crash (it will now print a number to standard output) while maintaining the same behavior as the original program for all other inputs.

*Note that the challenges do not necessarily represent the best protections available from the obfuscation tools we have at our disposal - rather, they were designed to be possible to crack. Our end goal with this competition is to collect information for the community on how attacks are carried out in the real world. In other words, this is not a competition between providers of obfuscation tools, but a competition between attackers! We do not provide information on which tool was used to protect which challenge, and which protective transformations were employed - figuring this out is part of the challenge.*

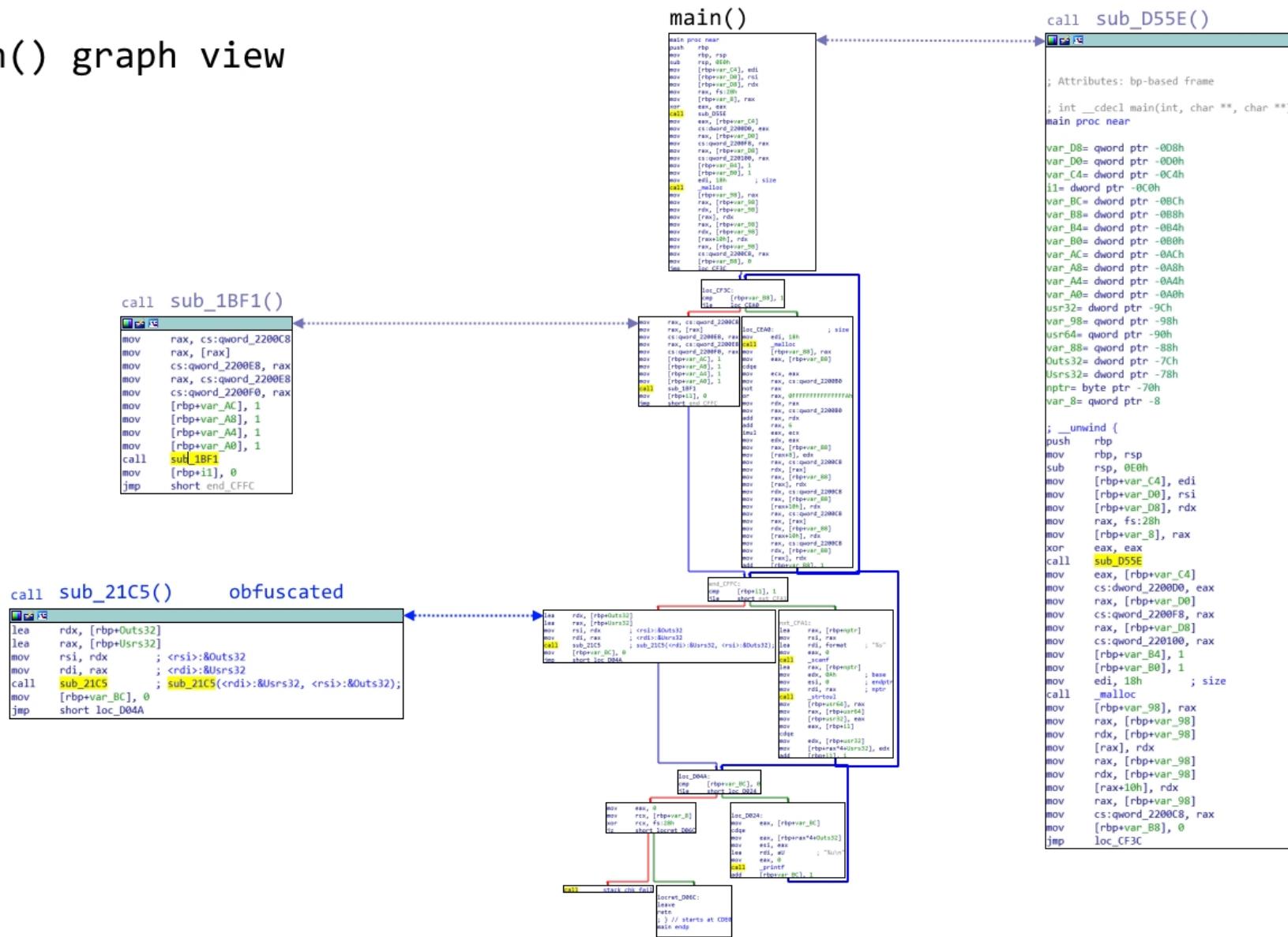


## Round 2

- **Challenge 8 ( VERY LIGHT EXTRACT ARM )**: challenge-8-x86-Linux.exe takes a license key as its first argument on the command line. It is your job to find this key. The program prints an error message if the key is wrong.
- **Challenge 9 ( MEDIUM EXTRACT ARM )**: challenge-9-x86-Linux.exe is a protected version of the bsdtar program provided by libarchive. A list of command-line arguments can be generated with ./bsdtar --help. An additional, undocumented argument (-K <some string>) can be used to pass a license key to the program. Valid tarballs are produced only when this key is correct. Multiple license keys are possible. It is your job to write a key generator with which correct license keys can be generated. To generate a tarball use this command: ./bsdtar --create --gzip -f tarball.tar.gz -K \$KEY \$INPUT\_DIR. To list the files in the tarball use this command: ./bsdtar --list -f tarball.tar.gz -K \$KEY. This command will fail if the tarball is corrupted.
- **Challenge 10 ( LIGHT TAMPER X86 )**: You are given a binary challenge-10-x86-Linux.exe whose first argument is a password and whose second argument is the input to a hash function. The output is a hash of the second. Your task is to remove the password check from the binary. The cracked binary should take 1 argument and function identically to the original binary if the correct password was entered. Submit the cracked binary.

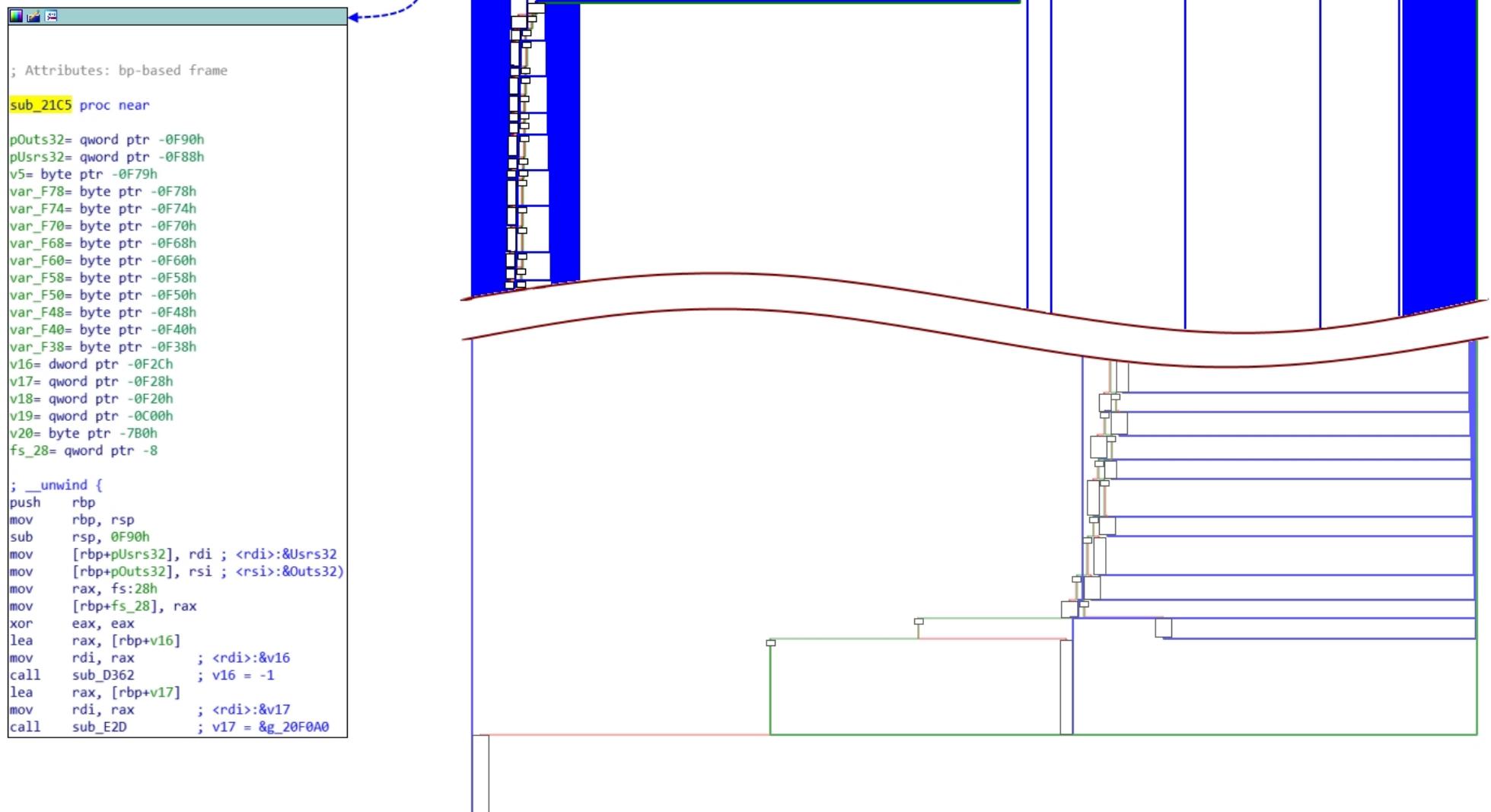
- —
- —
- —

main() graph LR



Pic. 2 Graph view of main function

## sub\_21C5() graph view



Pic. 3 Graph view of `sub_21c5` subroutine

## Шаг 2.1. Анализ структуры псевдокода.

Псевдокод состоит из 2-х частей: **инициализирующей** (init) и **исполняющей** (loop). В инициализирующей части процедуры sub\_D362(), sub\_E2D() устанавливают начальные значения служебных переменных (set values). В исполняющей части процедура sub\_9E5() выбирает идентификатор блока (get id), которому передается управление. Выбранный блок либо сам (inline type) выполняет целевые действия, либо вызывает подчиненные процедуры (function type), которые их выполняют (execute id). Обновляются значения служебных переменных для следующей итерации выбора идентификатора блока (update values). Один из блоков при его выборе осуществляет завершение работы исполняющей части ([Pic. 4](#)).

Таким образом, **псевдокод** является **интерпретатором байт-кода**, в который был транслирован **исходный код** программы.

## Шаг 2.2. Анализ переменных псевдокода.

К **глобальным** переменным относятся:

- [g\\_20F0A0](#) (table) массив байт-кода

К переменным **состояния** относятся:

- [v5](#) (id) идентификатор блока

- [v18](#), [v19](#) (value) массивы хранимых значений

- [v20](#) (point) массив точек передачи управления блокам, из которых используется только точка старта (entry point)

- tmp (value) хранимые значения, обращение к которым осуществляется через указатели, рассчитанные относительно v20

К **служебным** переменным относятся:

- [v16](#) (index) индекс элемента из массива v18

- [v17](#) (table) указатель на элемент из массива g\_20F0A0

- [vF40](#), ..., [vF80](#) (tmp) временные элементы

Хранимые **значения** и элементы из массива байт-кода **интерпретируются по-разному в зависимости от кода блока**.

## sub\_21C5() pseudo code

```

1 unsigned __int64 __fastcall sub_21C5(__int64 a1, __int64 a2)
{
    __int64 pOuts32; // [rsp+0h] [rbp-F90h]
    __int64 pUsrs32; // [rsp+8h] [rbp-F88h]
    signed int *v17; // [rsp+68h] [rbp-F28h]
    __int64 v18[100]; // [rsp+70h] [rbp-F20h]
    __int64 v19[138]; // [rsp+390h] [rbp-C00h]
    char v20[1960]; // [rsp+7E0h] [rbp-7B0h]
    unsigned __int64 fs_28; // [rsp+F88h] [rbp-8h]

    pUsrs32 = a1;
    pOuts32 = a2;
    fs_28 = __readfsqword(0x28u);
    sub_D362(&v16);
    sub_E2D(&v17);

    init
    loop
    while ( 1 )
    {
        while ( 1 )
        {
            ...
            while ( 1 )
            {
                while ( 1 )
                {
                    sub_9E5((__BYTE **)&v17, &v5);
                    if ( v5 != 114 )
                        break;
                    sub_D5D9((__int64)v18, &v16, (__int64)v19, &v17);
                    sub_E46(&v16, &v17);
                    if ( v5 != 35 )
                        break;
                    sub_1F4B(&v17);
                    sub_1AFE(v18, &v16);
                    if ( v5 != 108 )
                        break;
                    ...
                    {
                        v17 = (signed int*)((char *)v17 + 1);
                        v19[*v17] = *(_QWORD*)(v17 + 1);
                        v18[v16 + 1] = *(_QWORD*)(v17 + 3);
                        v19[v17[5]] = v18[v16 + 1];
                        v17 += 6;
                    }
                    break;
                }
            }
            v17 = (signed int*)((char *)v17 + 1);
            return __readfsqword(0x28u) ^ fs_28;
        }
    }
1868 }

```

**set values**

```

__DWORD * __fastcall sub_D362(__DWORD *a1)
{
    __DWORD *result; // rax
    result = a1;
    *a1 = -1;
    return result;
}
int v16 = -1; // index

```

**get id**

```

__BYTE * __fastcall sub_9E5(__BYTE **a1, __BYTE *a2)
{
    __BYTE *result; // rax
    result = a2;
    *a2 = **a1;
    return result;
}
char v5 = *((char*)v17); // id

```

**execute id**

```

__int64 __fastcall sub_D5D9(__int64 a1, __DWORD *a2, __int64 a3, signed int **a4)
{
    __int64 result; // rax
    *a4 = (signed int*)((char *)a4 + 1);
    result = a1;
    *(_QWORD*)(a1 + 8LL * (*a2 + 1)) = *(_QWORD*)(a3 + 8LL * (*a4)[1])
                                            + *(_QWORD*)(a3 + 8LL * **a4);
    return result;
}

```

**update values**

```

__QWORD * __fastcall sub_E46(__DWORD *a1, __QWORD *a2)
{
    __QWORD *result; // rax
    ++*a1;
    result = a2;
    *result += 8LL;
    return result;
}
v16 = v16 + 1;
v17 = (char*)v17 + 8;

```

Pic. 4 Pseudo code of sub\_21c5 subroutine

## Шаг 2.3. Ручная часть создания кода.

В файле **types.h** определим необходимые **типы** данных. Тип объединение (**mem\_t**) используется для получения доступа к одним и тем же данным через указатели разных типов. Тип структура (**obj\_t**) используется для хранения всех служебных переменных и переменных состояния. Остальные типы относятся к собственному функционалу создаваемого кода. В файле **defs.h** введем необходимые **псевдонимы**, совпадающие с именами переменных из псевдокода. В файле **vars\_g4c.h** зададим значения **глобальных** переменных.

Массив **байт-кода** создадим, используя шестнадцатеричный редактор (**Winhex**) ([Pic. 5](#)). В файле **test.cpp** создадим функцию **\_t\_test\_()**, которая повторяет структуру псевдокода. Создадим функции **sub\_9E5()**, **sub\_D362()**, **sub\_E2D()**, используя их псевдокод из декомпилятора ([Pic. 7](#)).

## Шаг 2.4. Автоматическая часть создания кода.

Скрипты на языке python используют простой **текстовый поиск и замену** подстрок. Для получения конечного результата они запускаются последовательно:

1) `t1_gh2io.py --parameters='input-file=./input/g_21c5.txt;log-file=./log/t1_g_21c5.txt'`

**Входные** данные это сгенерированный **псевдокод** процедуры **sub\_21C5()**, обработанный предварительно ручной заменой для удаления ненужного текста.

**Выходные** данные это **файл псевдокода** (`./tmp/t1_21c5.txt`), в котором сделаны замены указателей на использование индексов для обращения к элементам массивов.

Пример:

```
* (v18 + (v16 + 1)) = -* (v19 + p17[0]);    ->    v18[v16+1]=-v19[p17[0]];
```

2) `t2_gh2io.py --parameters='input-file=./tmp/t1_21c5.txt;log-file=./log/t2_g_21c5.txt'`

**Входные** данные это **псевдокод** из п.1.

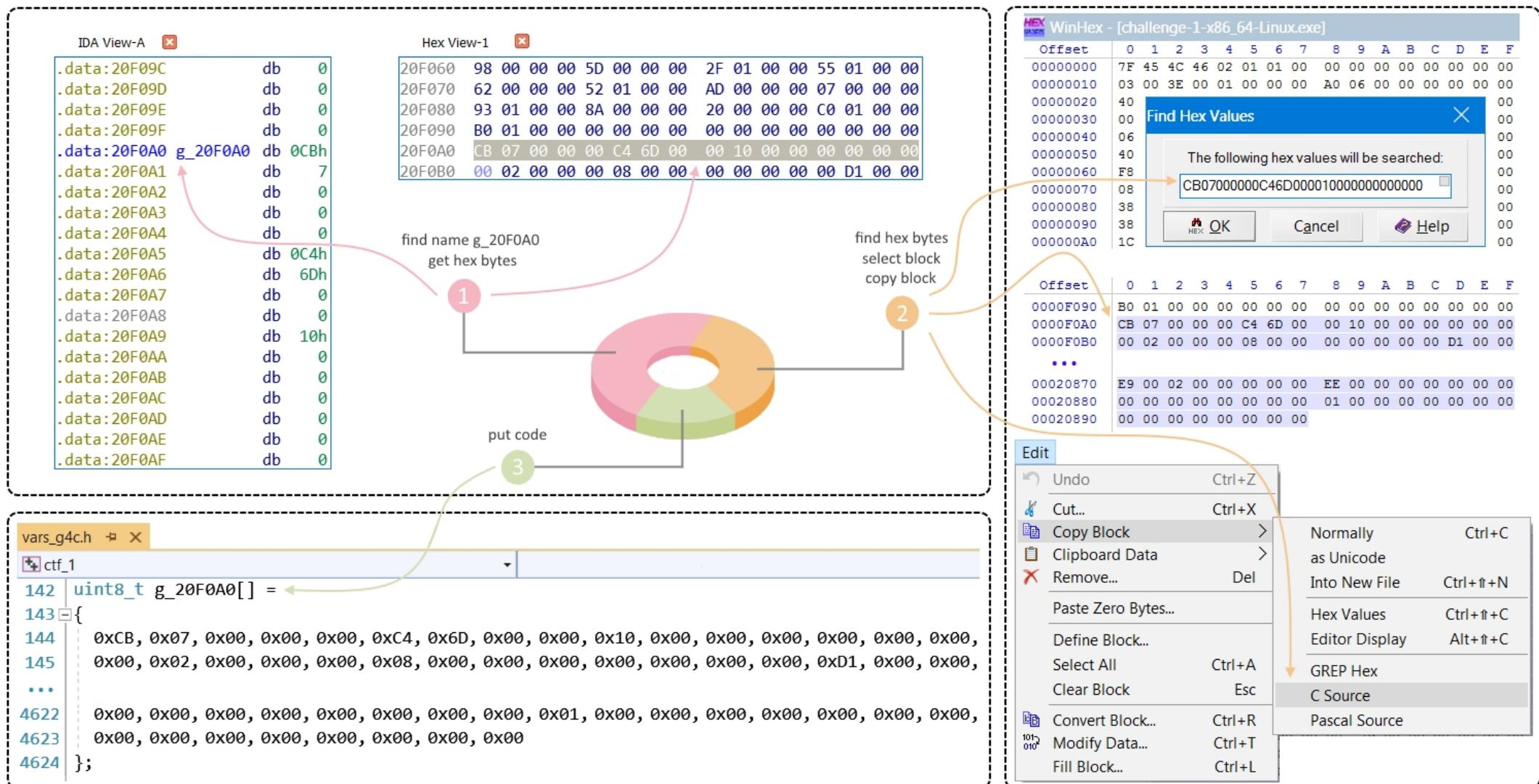
**Выходные** данные это **файл псевдокода** (`./tmp/t2_21c5.txt`), в котором сделаны замены действий на вызов соответствующих функций.

Пример:

```
v18[v16 + 1] = v19[p17[0]];    ->    cp_1819(o, _v16 + 1, 0);
```

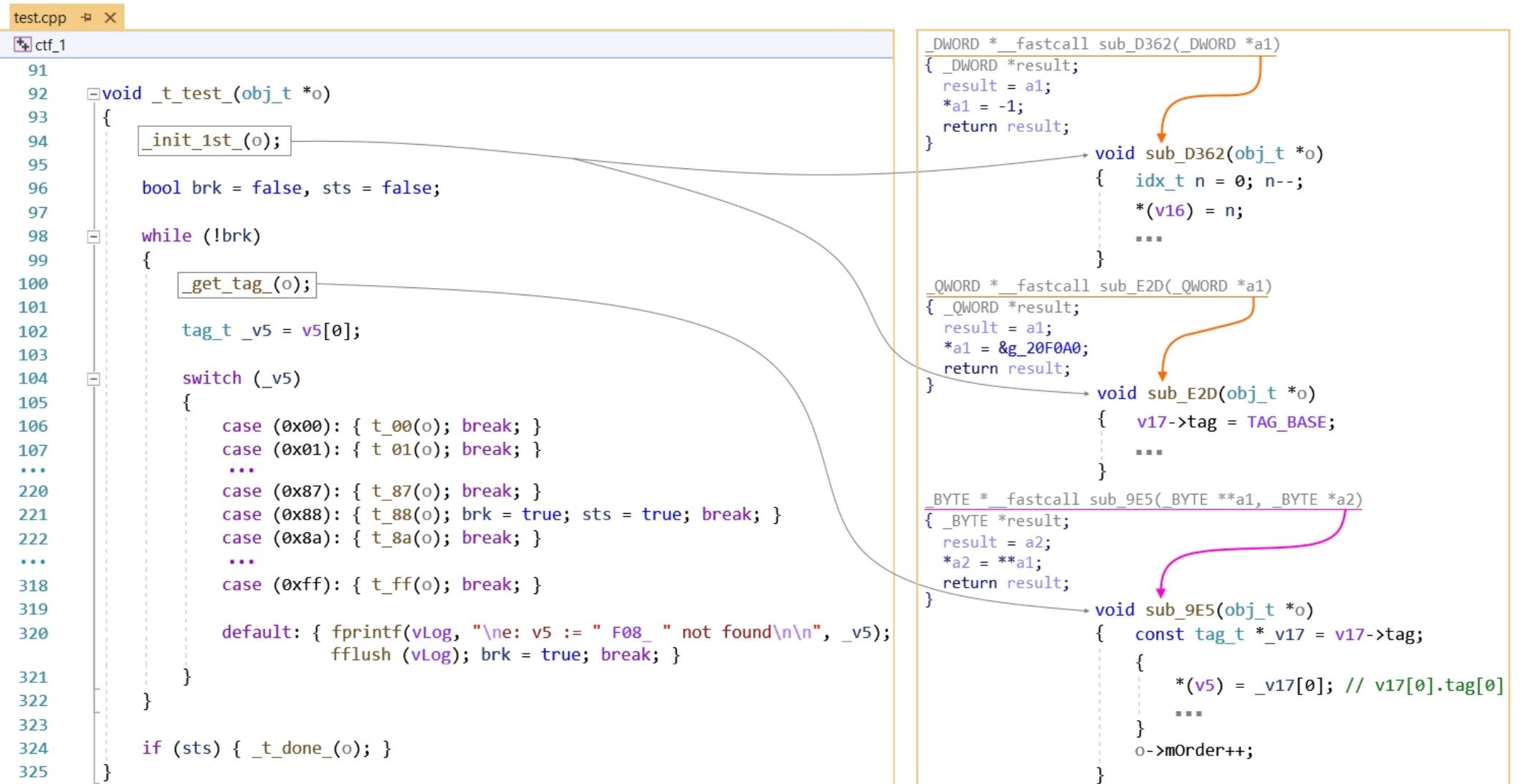
```
v19[p17[1]] = (g_20F0A0 + p17[2]);    ->    ad_glb_19(o, 1, 2);
```

## make g\_20F0A0[]



Pic. 5 Auto make c code of g\_20f0a0 byte array

## make \_t\_test\_()



Pic. 7 Make c code of \_t\_test\_ function

3) `t3_gh2io.py --parameters='input-file=./tmp/t2_21c5.txt;log-file=./log/t3_g_21c5.txt'`

**Входные** данные это **псевдокод** из п.2.

**Выходные** данные это **файлы кода на языке си** (`./output/t_<id>.txt`), каждый из которых реализует функцию `t_<id>()` для блока с идентификатором (`id`).

Пример для файла `t_0a.txt`:

```
void t_0a(obj_t *o)
{
    mv_tag_17(o, 1); // v17 += 1;
    {
        const idx_t *_v17 = v17->idx, _v16 = *(v16);
        seq_181819(o, _v16, _v16, 0);
    }
    mv_idx_17(o, 5); // (idx_t) v17 += 5; (tag_t) v17=v17+5;
}
```

4) `t4_gh2io.py --parameters='input-dir=./input;log-file=./log/t4_g_21c5.txt'`

**Необязательный** служебный скрипт, проверяющий имена файлов из п.3.

5) `w1_gh2io.py --parameters='input-file=./input/g_fun_.txt;log-file=./log/t1_fun_.txt'`

**Входные** данные это сгенерированный **псевдокод для всех функций**, которые **вызываются** из процедуры `sub_21C5()`, обработанный предварительно ручной заменой для удаления ненужного текста.

**Выходные** данные это **файл псевдокода** (`./tmp/t1_fun_.txt`), который аналогичен п.1.

6) `t2_gh2io.py --parameters='input-file=./tmp/t1_fun_.txt;log-file=./log/t2_fun_.txt'`

**Входные** данные это **псевдокод** из п.5.

**Выходные** данные это **файл псевдокода** (`./tmp/t2_21c5.txt`), который аналогичен п.2.

7) `w3_gh2io.py --parameters='input-file=./tmp/t2_fun_.txt;log-file=./log/t3_fun_.txt'`

**Входные** данные это **псевдокод** из п.6.

**Выходные** данные это **файл кода на языке си** (`./output/t3_fun_.txt`), который реализует функции.

Пример:

```
void fun_0010d5d9(obj_t *o)
{
    v17 = v17 + 1;
    v18[ v16 + 1 ] = v19[ p17[ 0 ] ] + v19[ p17[ 1 ] ];
}

void fun_0010d5d9(obj_t *o)
{
    mv_idx_17(o, 1); // (idx_t) v17 += 1; (tag_t) v17=v17+1;
    add_181919(o, _v16 + 1, 0, 1);
}
```

Добавим весь созданный код в решение. Для выполнения его построения в среде разработки устраним возникшие ошибки ([Pic. 8](#)).

Замечания:

- предварительная ручная замена по удалению ненужного текста не реализована с использованием скриптов
- не учтены модификаторы размеров переменных (8, 32, 64 бита) и их знакового расширения, что приводит к дополнительному объемному анализу
- не для всех числовых индексов введены макроопределения, что исключает применение к ним собственных типов данных

Для реализации возможности задания **размера переменных** и учета их **знакового расширения** определим **битовые маски** (**bm\_t**) и функции работы с ними.

Пример:

```
// беззнаковые
bm_t bm_32(uint32_t n);
bm_t bm_8(uint32_t n);

// знаковые
bm_t bm_s32(uint32_t n);
bm_t bm_s8(uint32_t n);
```

1 copy text and replace text

2 replace pseudocode and make code

3 make solution

**g\_21c5.txt**

```

void FUN_001021c5(undefined8 param_1,undefined8 param_2)
{
long in_FS_OFFSET;
...
local_f10 = *(in_FS_OFFSET + 0x28);
local_f98 = param_2;
local_f90 = param_1;
FUN_0010d362(&v16);
FUN_00100e2d(&v17);
while( true ) {
while( true ) {
...
if (v5 != 0xc9) break;
v19[p17[0]] = v18[v16];
v18[v16 - 1] = v19[p17[1]];
v16 = v16 - 2;
v17 = v17 + 9;
}
if (v5 != 0xf3) break;
v18[v16] = (v19[p17[0]] % v18[v16]);
v17 = v17 + 5;
}
...
return;
}
...

```

**Project**

- Gh2io
  - log
  - input
    - g\_21c5.txt
    - g\_fun\_txt
  - tmp
    - t1\_21c5.txt
    - t2\_21c5.txt
    - t1\_fun\_txt
    - t2\_fun\_txt
  - output
    - t\_00.txt
    - t\_01.txt
    - t\_02.txt
    - t\_03.txt
    - ...
    - t\_fe.txt
    - t\_ff.txt
    - t3\_fun\_txt
- Run/Debug Configurations

**Python**

- Inline type
  - c1 : clear
  - c2 : replace
  - c3 : gen
  - c4 : compare
- Function type
  - f1 : f\_clear
  - f2 : f\_replace
  - f3 : f\_gen

**Configuration:** c1 : clear

**Script path:** \Gh2io\t1\_gh2io.py

**Parameters:** --parameters='input-file=./input/g\_21c5.txt;+'

**Environment**

**Environment vars:** PYTHONUNBUFFERED=1

**Python interpreter:** Project Default (Python 3.8 (gh2io))

**Interpreter options:**

**Working directory:** \Gh2io

**Решение "ctf\_1" (проект: 1 из 1)**

- ctf\_1
  - Ссылки
  - Внешние зависимости
  - tags
    - t\_00.cpp
    - t\_01.cpp
    - t\_02.cpp
  - t\_02.cpp
 

```

void t_02(obj_t *o)
{
    set_tag_17(o, 1); // (tag_t*) v17 += 1;
    ...
    cpy_1918(o, 0, STP_20);
    set_val_18(o, STP_20, 1);
    cpy_1918(o, 3, STP_20);
    mul_1919(o, 5, 6, 4);
    cpy_1819(o, STP_20, 7);
    set_val_19(o, 16, 8);
    add_181819(o, STP_20, 11, STP_20);
    ...
    // TODO: S32 -> 64
    // mov    rcc, [rbp + rax * 8 + v19]
    // mov    eax, word ptr [rbp + rax * 8 + v18]
    // movsd  rdx, eax
    // add    rdx, rcc
    // mov[rbp + rax * 8 + v18], rdx
    add_181819(o, STP_20, bm_s32(STP_20), 12);
    cpy_1918(o, 13, STP_20);
    ...
    wrt64_1919(o, 14, 15);
}
set_idx_16(o, STP_01); // (Idx_t) v16 -= 1;
set_idx_17(o, 16); // (Idx_t) v17 += 16; (tag_t*) v17=v17+0x41;
```
  - t4c.h
  - main.cpp
  - tags.cpp
  - tags.h
  - test.cpp
  - test.h
  - tfns.cpp
 

```

void fun_0010d5d9(obj_t *o)
{
    set_tag_17(o, 1); // (tag_t*) v17 += 1;
    add_181919(o, STP_P1, 0, 1);
}
```
  - vars.h
  - vars\_g4c.h

Обозреватель решений

Pic. 8 Make c code of ctf\_1 solution

Установим по умолчанию размер переменных **64 бита**. По характерным **ключевым** словам: **SLOODWORD**, **LODWORD**, **LOBYTE** в псевдокоде Idapro найдем (подстрока **LO**) и исправим места, которые требуют использования битовой маски (**32-бита, 8-бит**).

Пример:

```
if ( v5 != 65 )
    break;
LODWORD(v19[*v17]) = v19[v17[1]] & LODWORD(v18[v16--]);  <->  and_191819(o, bm_32(0), bm_32(STP_Z0), bm_32(1));
```

Создадим настраиваемые функции тестирования и протоколирования выполнения байт-кода.

### Шаг 3. Реализация байт-кода.

Требуется создать код на языке си, реализующий интерпретируемый байт-код.

Зададим **минимальные настройки** протоколирования (small) и создадим **протокол** выполнения байт-кода, определив 2 пользовательских числа.

#### Шаг 3.1. Анализ структуры протокола выполнения байт-кода.

Протокол состоит из 2-х частей: **инициализирующей** (init) и **исполняющей** (exec). В **инициализирующей** части заполняется **таблица** точек передачи **управления** блокам, из которых **используется** только **точка старта** (entry point). В **исполняющей** части **вычисляется** значение **результата**. **Маркер** протокола (**s:**) означает изменение порядка выполнения байт-кода, которое соответствует передаче управления или **ветвлению** в исходном коде. Начало исполняющей части следует за **的独特ной** строкой:

```
s: v17 := v19[ 5 ] =
```

**Длина** исполняющей части **меняется** в зависимости от вводимых чисел.

Таким образом, для восстановления **полного** исходного **кода** программы потребуется вводить пользовательские **числа**, обеспечивая исполнение **всех** его **веток**.

## Шаг 3.2. Анализ переменных протокола.

В **протоколе** отражаются **значения и адреса** переменных, которые содержат вводимые значения, **результат**, рабочие флаги и массивы. Инициализирующая часть не влияет на результат. Поэтому может не рассматриваться. Адреса переменных **меняются** для каждого протокола.

Python скрипт выполняет поиск и замену меняющихся значений.

1) `z52_vvv.py --parameters='input-file=./input/log_am3.txt;log-file=./log/z52_vvv.txt'`

**Входные** данные это исполняющая часть **протокола** и заданные в скрипте значения **адресов и имен переменных**. Часть адресов (`user, actn`) может быть найдена в начале исполняющей части. Другая часть (`calc`) может быть найдена после выполнения предыдущей замены.

Пример:

```
l_items.append(((13FD391E8, (2, 4)), 'pRslt'))  
l_items.append(((13FD391E0, (2, 4)), 'pUser'))  
l_items.append(((27F223, (10, 4)), 'pActn'))  
l_items.append(((13fd39230, (10, 4)), 'pCalc'))
```

**Выходные** данные это **файл протокола** (`./output/log_am3.txt`), который не зависит от значений **адресов** и части **значений переменных** (Pic. 9).

## Шаг 3.3. Создание кода.

Создание кода состоит из следующих действий:

- задаем 2 числа
- создаем и обрабатываем протокол
- создаем код, реализующий вычисления из протокола
- отмечаем все полученные ветвления кода
- проверяем правильность результата
- выполняем поиск новых чисел для одного из отмеченных ветвлений
- повторяем все действия

Поиск новых чисел выполняется сканнером.

Код сканнера аналогичен создаваемому коду, в который добавлены **ускоряющие** проверки и прерывание по условию (Pic. 10).

1 run solution and make log

analyze log and discard useless

3

find ptrs and define names

Решение "ctf\_1" (проекты: 1 из 1)

ctf\_1

Ссылки

main.cpp

```
1 int main(int argc, char* argv[])
2 {
3     test_t ts[] = { user input
4         .p = "log_am3.txt",
5         .u0 = 0xDC021,
6         .u1 = 0x5B6FA
7     };
8     uint32_t nt = n_count(ts);
9     for (uint32_t n = 0; n < nt; ++n)
10    {
11        t_test(&ts[n]);
12    }
13    return 0;
14 }
```

vars\_g4c.h

```
1 /* small
2  */
3 opts_t g_opts = {
4     .tg = false,
5     .vf = true, control flow
6     .adp = { .a = false, .e20 = false, ... },
7     .rd = false,
8     .wr = true, write value
9     .cp = false,
10    .get = { .a = false, .e16 = false, ... },
11    .set = { .a = false, .e16 = false, ... }
12 };
13
14 uint8_t g_20F040[] = bytecode
15 {
16     0xCB, 0x97, 0x00, 0x00, 0x00, 0xC4, ...
17     0x00, 0x02, 0x00, 0x00, 0x00, 0x00, ...
18     0x00, 0x00, 0x00, 0x00, 0x00, 0x01, ...
19     0xE9, 0x00, 0x02, 0x00, 0x00, 0x00, ...
20     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
21     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
22     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
23     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
24     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
25     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
26     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
27     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
28     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
29     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
30     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
31     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
32 }
```

Обозреватель решений

log\_am3.txt

```
1 init i: v16 : = -1
2 i: v17[ 0] : = BASE = 13FD24060
3 *: v19[ 3] := v19[ 2] * v19[ 1] = 8 * D1 = 688
4 +: v19[ 5] := v18[ 1] + v18[ 0] = 0 + 688 = 688
5 +: v18[ 0] := v19[ 0] + v18[ 0]:S32 = 27EB87 + 688 = 27F20F
6 w: *(v19[ 6]) := v19[ 7] ; *(27F20F) := 13FD2AE24
...
23 *: v18[ 0] := v19[ 0] * v19[ 1] = CC * 8 = 660
24 +: v18[ 0] := v18[ 0] + v19[ 2] = 660 + 0 = 660
25 +: v19[ 3] := v18[ 0]:S32 + v18[ 1] = 660 + 27EB87 = 27F1E7
26 w: *(v18[ 0]) := v19[ 4] ; *(27F1E7) := 13FD2BB77 entry point
...
839 *: v18[ 0] := v19[ 1] * v19[ 2] = 0 * 8 = 0
840 +: v18[ 0] := v19[ 3] + v18[ 0] = 0 + 0 = 0
841 +: v18[ 0] := v18[ 0]:S32 + v19[ 0] = 0 + 27EB87 = 27EB87
842 w: *(v19[ 4]) := v19[ 5] ; *(27EB87) := 13FD341FC
```

goto

Search for: S:

Find All

Find All In All Tabs

Find All In Folder

Line 5 matches

```
843 w: *(v18[ 0]) := v18[ 1] ; *(27
844 -: v18[ 0] := v19[ 1] - v18[ 0]
845 *: v18[ 0] := v19[ 2] * v19[ 3]
846 +: v18[ 0] := v18[ 0] + v19[ 4]
847 +: v18[ 0] := v19[ 0] + v18[ 0]
848 S: v17 := v19[ 5] = 13FD2BB77
1082 S: VF70[ 0] := 0
1083 S: VF68[ 0] := 1
1084 S: VF80[ 0]:32 := 30
1085 S: VF60 := 13FD330E7
```

exec

```
849 w: *(v19[ 0]):32 := v19[ 1]:32 ; *(27F27B) := 0
850 w: *(v19[ 0]) := v19[ 3] ; *(27F2BF) := 13FD391E8
851 *: v18[ 0] := v19[ F] * v19[ 10] = 1 * 4 = 4
852 +: v19[ 11] := v18[ 0] + v19[ D] = 4 + 13FD391E0 = 13FD391E4
853 *: v19[ 9] := v19[ 8] * v19[ 7] = 4 * 0 = 0
854 +: v19[ A] := v18[ 0] + v19[ 6] = 0 + 13FD391E0 = 13FD391E0
855 |: v19[ 13]:32 := v19[ 12] | v19[ B] = DC021 | 5B6FA = DF6FB
856 *: v18[ 1] := v19[ 1] * v19[ 2] = 0 * 4 = 0
857 +: v19[ 4] := v18[ 2] + v18[ 1] = 0 + 0 = 0
858 +: v18[ 1] := v19[ 0] + v18[ 1]:S32 = 27F223 + 0 = 27F223
859 w: *(v18[ 1]):32 := v18[ 0]:32 ; *(27F223) := DF6FB
...
1617 *: v18[ 1] := v19[ 3] * v19[ 4] = 0 * 4 = 0
1618 +: v19[ 5] := v18[ 1] + v19[ 1] = 0 + 13FD391E8 = 13FD391E8
1619 w: *(v18[ 1]):32 := v18[ 0]:32 ; *(13FD391E8) := CC800720
```

log\_am3.txt

```
1 i: v16 : = -1
2 i: v17[ 0] : = BASE = 13FD24060
3 exec
4 w: *(v19[ 0]):32 := v19[ 1]:32 ; *(27F27B) := 0 pRslt
5 w: *(v19[ 0]) := v19[ 3] ; *(27F2BF) := 13FD391E8 pUser1
6 *: v18[ 0] := v19[ F] * v19[ 10] = 1 * 4 = 4 pUser0
7 +: v19[ 11] := v18[ 0] + v19[ D] = 4 + 13FD391E0 = 13FD391E4 pUser1
8 *: v19[ 9] := v19[ 8] * v19[ 7] = 4 * 0 = 0 pUser0
9 +: v19[ A] := v18[ 0] + v19[ 6] = 0 + 13FD391E0 = 13FD391E0 pUser0
10 |: v19[ 13]:32 := v19[ 12]:32 | v19[ B]:32 = DC021 | 5B6FA = DF6FB
11 *: v18[ 1] := v19[ 1] * v19[ 2] = 0 * 4 = 0 User1 User0
12 +: v19[ 4] := v18[ 2] + v18[ 1] = 0 + 0 = 0 pActn0
13 +: v18[ 1] := v19[ 0] + v18[ 1]:S32 = 27F223 + 0 = 27F223 pActn0
14 w: *(v18[ 1]):32 := v18[ 0]:32 ; *(27F223) := DF6FB Actn0
...
139 *: v19[ 17] := v19[ 16] * v19[ E] = 4B286D12 * 2BF38EB3 = 87DA3F96
140 w: *(v18[ 0]):32 := v19[ 18]:32 ; *(27F247) := 87DA3F96 pActn9 Actn9
141
142 *: v19[ 2] := v19[ 1] * v19[ 0] = 4 * 0 = 0 pCalc0
143 +: v18[ 0] := v19[ 3] + v18[ 0] = 0 + 0 = 0
144 +: v19[ 4] := v18[ 0]:S32 + v18[ 1] = 0 + 13fd39230 = 13fd39230 pCalc0
145 *: v18[ 1] := v19[ 7] * v19[ 8] = 0 * 4 = 0
146 +: v18[ 1] := v19[ 5] + v19[ 9] = 0 + 13FD391E0 = 13FD391E0 pCalc0
147 *: v18[ 2] := v19[ d] * v19[ e] = 1 * 4 = 4
148 +: v19[ f] := v18[ 2] + v19[ b] = 4 + 13FD391E0 = 13FD391E4 pCalc0
149 *: v19[ 11]:32 := v18[ 2]:32 * v18[ 1]:32 = dc021 * 5b6fa = 94aa163a pCalc0
150 w: *(v18[ 0]):32 := v19[ 11]:32 ; *(13fd39230) := 94aa163a Calc0 pCalc0
...
249 +: v18[ 1] := v18[ 1]:S32 + v19[ 0] = 24 + 13fd39230 = 13FD39254
250 w: *(v18[ 1]):32 := v18[ 0]:32 ; *(13FD39254) := 38BB95EB Calc9
...
z52.vvv.py
l_items.append((13FD391E8, (2, 4)), 'pRslt')
l_items.append((13FD391E0, (2, 4)), 'pUser')
l_items.append((27F223, (10, 4)), 'pActn')
l_items.append((13fd39230, (10, 4)), 'pCalc')
...
1618 +: v19[ 5] := v18[ 1] + v19[ 1] = 0 + 13FD391E8 = 13FD391E8 pRslt
1619 w: *(v18[ 1]):32 := v18[ 0]:32 ; *(13FD391E8) := CC800720 Rslt
```

Pic. 9 Make c code of rst\_1 solution: first part

**4** [run script and replace ptrs](#)

**5** [analyze logs and make code](#)

**6** [make solution](#)

The screenshot shows a software interface with several panels:

- Project:** Shows a tree view of files and folders, including `Gh2io`, `log`, `tmp`, `input` (containing `log_am3.txt`), `output` (containing `log_am3.txt`), and various Python files like `__init__.py`, `i_args.py`, etc.
- Run/Debug Configurations:** A configuration panel for a Python script named `z52 : parser`. It includes fields for `Script path` (`\Gh2io\z52_vvv.py`), `Parameters` (`--parameters='input-file=./input/log_am3.txt;+<`), `Environment vars` (`PYTHONUNBUFFERED=1`), `Python interpreter` (`Project Default (Python 3.8 (gh2io))`), `Interpreter options`, and `Working directory` (`\Gh2io`). A callout points from the `log_am3.txt` file in the Project panel to the `Parameters` field.
- log\_am3.txt:** A window displaying assembly code with annotations. Annotations highlight memory addresses and variable names in green boxes. A yellow thinking emoji is positioned below the code.
- Solution Explorer:** Shows the `Rешение` (Solution) with files like `rst_1`, `actn.cpp`, `calc.cpp`, etc. A callout points from the `z52_vvv.py` code in the Run/Debug Configurations panel to the `rst_1` solution.
- Code Editor:** Shows the `main.cpp` file with C++ code corresponding to the assembly analysis. A callout points from the assembly code in the `log_am3.txt` window to the `main.cpp` code.

Pic. 10 Make c code of `rst_1` solution: second part

Замечания:

- составим таблицу возможных вариантов, используя **кодирование ветвлений**
- добавлять варианты будем до тех пор, пока не найдем **причину**, вызывающую **аварийный останов** программы

Таблица возможных вариантов ветвлений:

**Выполнено** (пользовательские числа найдены) :

```
AM = 3
AM = 2
AM = 1, FM = 2
AM = 1, FM = 0, G = 0
AM = 1, FM = 0, G = 1, Flag = 0
AM = 1, FM = 0, G = 1, Flag = 1 <- to access violation
AM = 0, CM = 3, Flag = 0
AM = 0, CM = 1, CM1 = 0, SIX = 1 ... 5
AM = 0, CM = 1, CM1 = 0, SIX = 0, G6 = 0
AM = 0, CM = 1, CM1 = 0, SIX = 0, G6 = 1, B14 = 0 ... 1
AM = 0, CM = 0
```

**Проверено** (пользовательские числа не найдены, но выполнение кода уже реализовано) :

```
AM = 1, FM = 3
AM = 1, FM = 1
AM = 0, CM = 2, Flag = 0
AM = 0, CM = 1, CM1 = 1, SIX = ...
```

**Никогда** не происходит:

```
AM = 0, CM = 3, Flag = 1
AM = 0, CM = 2, Flag = 1
```

**Не** исследовано:

```
AM = 0, CM = 1, CM1 = 2 ... 3
```

## Часть 2. Поиск решения.

### Шаг 1. Анализ восстановленного исходного кода.

В реализации варианта:

```
AM = 1, FM = 0, G = 1, Flag = 1 <- to access violation
```

присутствует код:

```
{
    if (Flag) { p_Rslt = nullptr; } // <- to access violation
}
```

В этом коде при наступлении определенного **условия (Flag)** указатель для записи результата (**p\_Rslt**) становится равным **nullptr**. Попытка **записи** результата по нулевому **указателю** приведет к **аварийному** останову программы. Поэтому данный вариант является искомым **решением**.

Замечания:

- условие является **составным**, что усложняет определение пользовательских чисел
- поиск пользовательских чисел выполняем **перебором** с использованием **оптимизированного** сканера, в котором отключены **замедляющие** циклы

### Шаг 2. Поиск пользовательских чисел.

В функции main() зададим настройки сканирования:

```
{ // AM = 1, FM = 0, G = 1, flag = 1 // access violation
.log = {.p = "log_AM1_FM0_G1_flag1.txt" },
.u0 = {.v = 0xffffffff, .e = 0x1 }, // диапазон сканирования
.ul = {.v = 0xffffffff, .e = 0x1 },
.ok = {.v = 1 }, // число искомых решений
.am = AM_1, // параметры ветвления
.fm = FM_0,
.g = g_and_1
}
```

Для **ускорения** поиска число **искомых** решений равно **1**.

Для ускорения поиска зададим **прерывания** по условиям, заданным в настройках.

В функции **actn\_run()**:

```
{  
    scan_t *s = Scanner;  
  
    if (m != s->am) { return; }  
}
```

В функции **func\_run()**:

```
{  
    scan_t *s = Scanner;  
  
    if (m != s->fm) { return; }  
}
```

В функции **\_fm0\_loop\_2\_()**:

```
{  
    scan_t *s = Scanner;  
  
    if (g != s->g) { return; }  
}
```

Для записи найденного решения в протокол зададим **критерий** поиска.

В функции **\_fm0\_g\_and\_1\_()**:

```
{ // speed up  
    scan_t *s = Scanner;  
  
    s->ok.b = (Actn[9] == 0xc4cfce5c); // <- критерий  
  
    if (s->ok.b)  
    { // :)  
        fprintf(s->log.h, " :: USER[0] = 0x%X (%u), USER[1] = 0x%X (%u)\n", User[0], User[0], User[1], User[1]);  
        fflush(s->log.h);  
    }  
  
    return;  
}
```

Файл протокола (./log\_AM1\_FM0\_G1\_flag1.txt) содержит найденные **пользовательские** числа.

Таким образом, **искомое** решение ([Pic. 11](#)):

```
USER[0] = 0xFFFFFFFFFB (4294967291)
USER[1] = 0xE6E1F222 (3873567266)
```

### Шаг 3. Выводы.

На примере решения этого задания была создана последовательность действий, которая позволила восстановить исходный код задания, выполнить его анализ и найти решение.

Последовательность действий **частично автоматизирована**. Однако, остается неформализованная **ручная работа**. Для дальнейшего **улучшения** последовательности действий требуется ее **проверка** работоспособности на **других** заданиях.

## 1 analyze solution

Решение "rst\_1"

- rst\_1
- Ссылки
- Внешние зависимости
- actn.cpp
- actn.h
- calc.cpp
- calc.h
- cnst.cpp
- cnst.h
- defs.h
- func.cpp

```
206 void _fm_g_and_1_(loop_t &h)
207 {
208     ub4 t b0 = { .u = Actn[9] },
209     b1 = { .u = 0 };
210     for (u32_t k = 0; k < ub4_n; ++k)
211     {
212         u32_t u = 0, v = 0, w = b0.b[k];
213         while (u < w)
214         {
215             u += 1; v += 1; w = b0.b[k];
216         }
217         b1.b[k] = u;
218     }
219     u32_t t1 = b1.u;
220     u32_t e1 = 0;
221     e1 = (e1 << (8 & 0x3f)) + 0xc4;
222     e1 = (e1 << (8 & 0x3f)) + 0xcf;
223     e1 = (e1 << (8 & 0x3f)) + 0xce;
224     e1 = (e1 << (8 & 0x3f)) + 0xc5;
225     Flag |= (t1 == e1); // 0xc4cfce5c
226     { // to access violation
227         if(flag){ p_Rslt=nullptr;
228             h.avFlag+=1; }
229     }
230 }
```

conditional setting the result pointer to zero

## 2 scan numbers

Решение "scn\_1"

- scn\_1
- Ссылки
- Внешние зависимости
- actn.cpp
- actn.h
- calc.cpp
- calc.h
- cnst.cpp
- cnst.h
- defs.h
- func.cpp
- func.h
- g4c.h
- main.cpp

```
13 int main(int argc, char* argv[])
14 {
15     scan_t ss[] =
16     {
17         ...
18         .log = { .p = "log_AM1_FM0_G1_flag1.txt" },
19         .u0 = { .v = 0xffffffff, .e = 0x1 },
20         .u1 = { .v = 0xffffffff, .e = 0x1 },
21         .ok = { .v = 1 },
22         .am = AM_1,
23         .fm = FM_0,
24         .g = g_and_1
25     };
26     // access violation
27     ...
28     u32_t ns = n_count(ss);
29     for (u32_t n = 0; n < ns; ++n)
30     {
31         Scanner = &ss[n];
32         s_scan();
33     }
34     return 0;
35 }
```



## 3 verify numbers

log\_AM1\_FM0\_G1\_flag1.txt

```
1 :: USER[0] = 0xFFFFFFF (4294967291), USER[1] = 0xE6E1F222 (3873567266)
```

```
@ubuntu:~/Downloads/ctf1$ ls -la
4096 .
4096 ..
133272 challenge-1-x86_64-Linux.exe

@ubuntu:~/Downloads/ctf1$ ./challenge-1-x86_64-Linux.exe
4294967291
3873567266
Segmentation fault (core dumped)

@ubuntu:~/Downloads/ctf1$
```

Pic. 11 Find an answer by scn\_1 solution