The Grand Reverse Engineering Challenge

This year, from May 21, **The Grand Reverse Engineering Challenge** was held, consisting of 2 rounds. In the 1st round of the competition, 7 tasks were offered for solving. Some of the tasks on the topic of deobfuscation require the restoration of the source code for their solution (**Pic.1**).

Our goal is to create a common sequence of actions that allows you to restore the source code for each task, analyze it and find a solution. (The goal is to deobfuscate the program and return it to idiomatic C)

Let's start solving the 1st task. By condition, it is required to enter two numbers that will cause an emergency shutdown of the program. (A successful submission consists of two 32-bit numbers that, when read from standard input, cause the program to crash with a segmentation fault)

Part 1. Restoring the source code.

Step 1. Preliminary analysis of the executable file in the decompiler.

It is required to find an obfuscated code, which is characterized by a large and confusing view of the executable graph. We will load and automatically analyze the downloaded executable file of the task into the Idapro (Ghidra) decompiler.

Let's find the **main()** function. Its executable graph is **small** (**Pic.2**). Three procedures are called: sub_D55E(), sub_1BF1(), sub_21C5() and several library functions that perform memory allocation, user input from the keyboard of 2 numbers and output the result.

Of all the procedures called, only **sub_21C5()** is obfuscated, since its executable graph is **large** and **confusing** (**Pic.3**).

Step 2. Implementation of the obfuscated code.

It is required to create a c code implementing an obfuscated procedure.

Let's generate the **pseudocode** of the procedure **sub_21C5()**. To analyze the structure and variables, we will generate it in **Idapro**, and for processing with scripts in **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.

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 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.

Participate

Submit

- **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 35329 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 maintaning the same behavior as the original program for all other inputs.

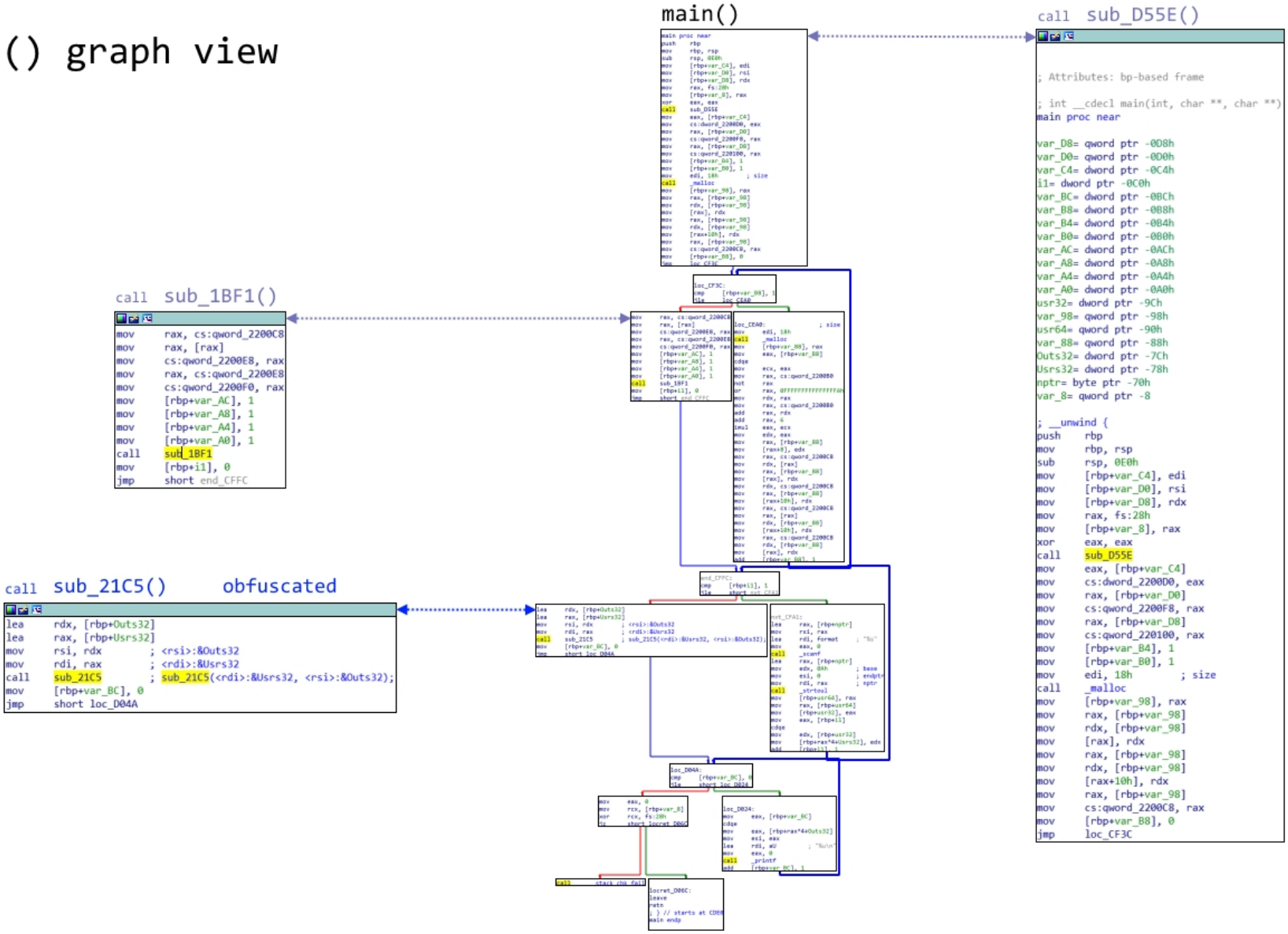
Round 2

- **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 ia the input to a hash function. The outoput 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.

Rules

Pic. 1 Grand Reverse Engineering Challenge

main() graph view



Pic. 2 Graph view of main function

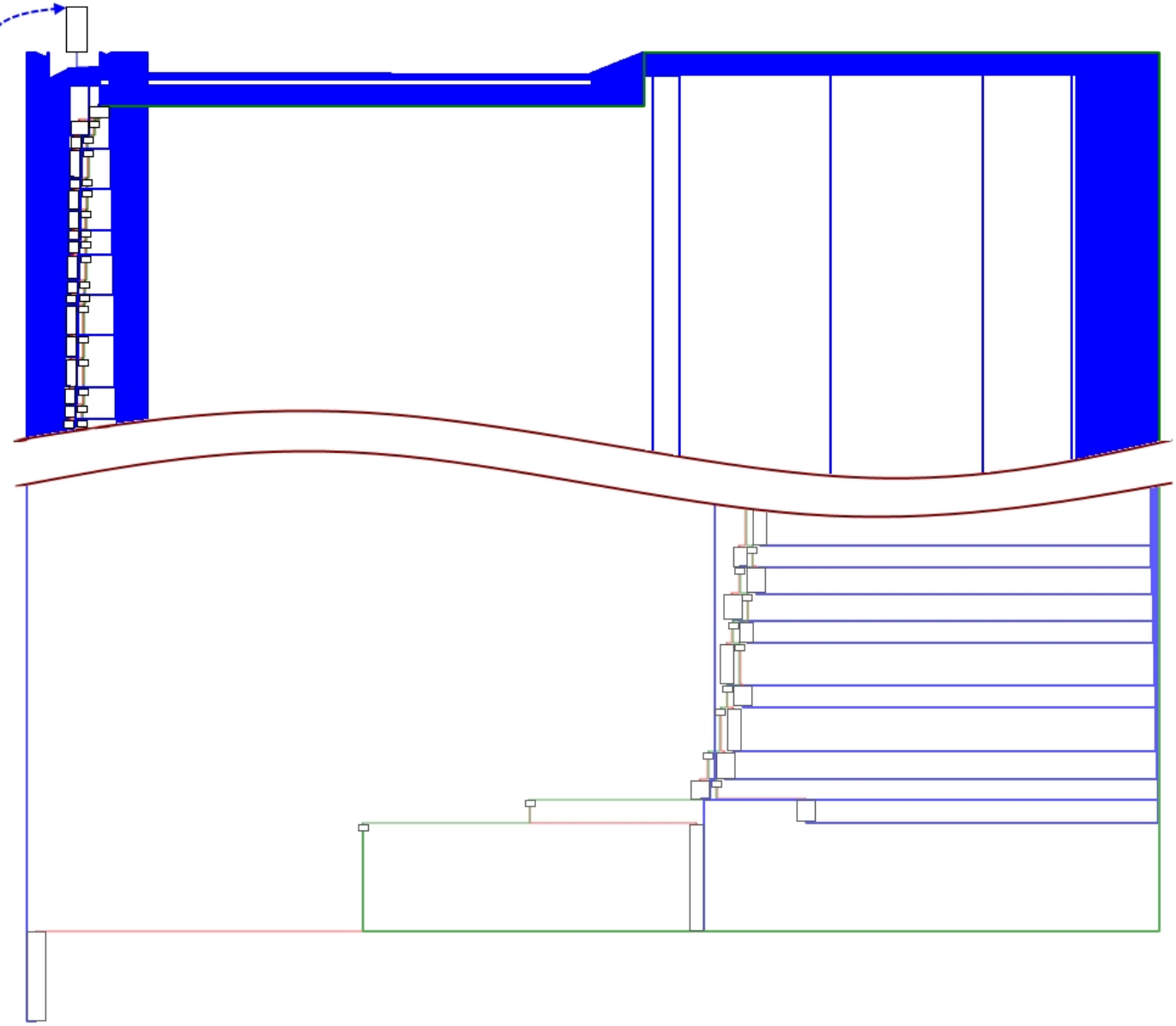
sub_21C5() graph view

```
; Attributes: bp-based frame

sub_21C5 proc near

pOuts32= qword ptr -0F90h
pUsrs32= qword ptr -0F88h
v5= byte ptr -0F79h
var_F78= byte ptr -0F78h
var_F74= byte ptr -0F74h
var_F70= byte ptr -0F70h
var_F68= byte ptr -0F68h
var_F60= byte ptr -0F60h
var_F58= byte ptr -0F58h
var_F50= byte ptr -0F50h
var_F48= byte ptr -0F48h
var_F40= byte ptr -0F40h
var_F38= byte ptr -0F38h
v16= dword ptr -0F2Ch
v17= qword ptr -0F28h
v18= qword ptr -0F20h
v19= qword ptr -0C00h
v20= byte ptr -7B0h
fs_28= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 0F90h
mov     [rbp+pUsrs32], rdi ; <rdi>:&Usrs32
mov     [rbp+pOuts32], rsi ; <rsi>:&Outs32)
mov     rax, fs:28h
mov     [rbp+fs_28], rax
xor     eax, eax
lea     rax, [rbp+v16]
mov     rdi, rax        ; <rdi>:&v16
call    sub_D362        ; v16 = -1
lea     rax, [rbp+v17]
mov     rdi, rax        ; <rdi>:&v17
call    sub_E2D         ; v17 = &g_20F0A0
```



Pic. 3 Graph view of sub_21c5 subroutine

Step 2.1. Pseudocode structure analysis.

The pseudocode consists of **2** parts: **initializing** (init) and **executing** (loop). In the initializing part of the procedure, sub_D362(), sub_E2D() set the initial values of service variables (set values). In the executing part, the sub_9E5() procedure selects the block identifier (get id) to which control is transferred. The selected block either performs the target actions itself (inline type), or calls subordinate procedures (function type) that perform them (execute id). The values of the service variables are updated for next iteration of the block ID selection (update values). One of the blocks, when selected, completes the work of the executing part (**Pic. 4**).

Thus, the **pseudocode** is an **interpreter** of the **bytecode** into which the **source code** of the program was translated.

Step 2.2. Analysis of pseudocode variables.

Global variables include:

- **g_**20F0A0 (table) byte code array

State variables include:

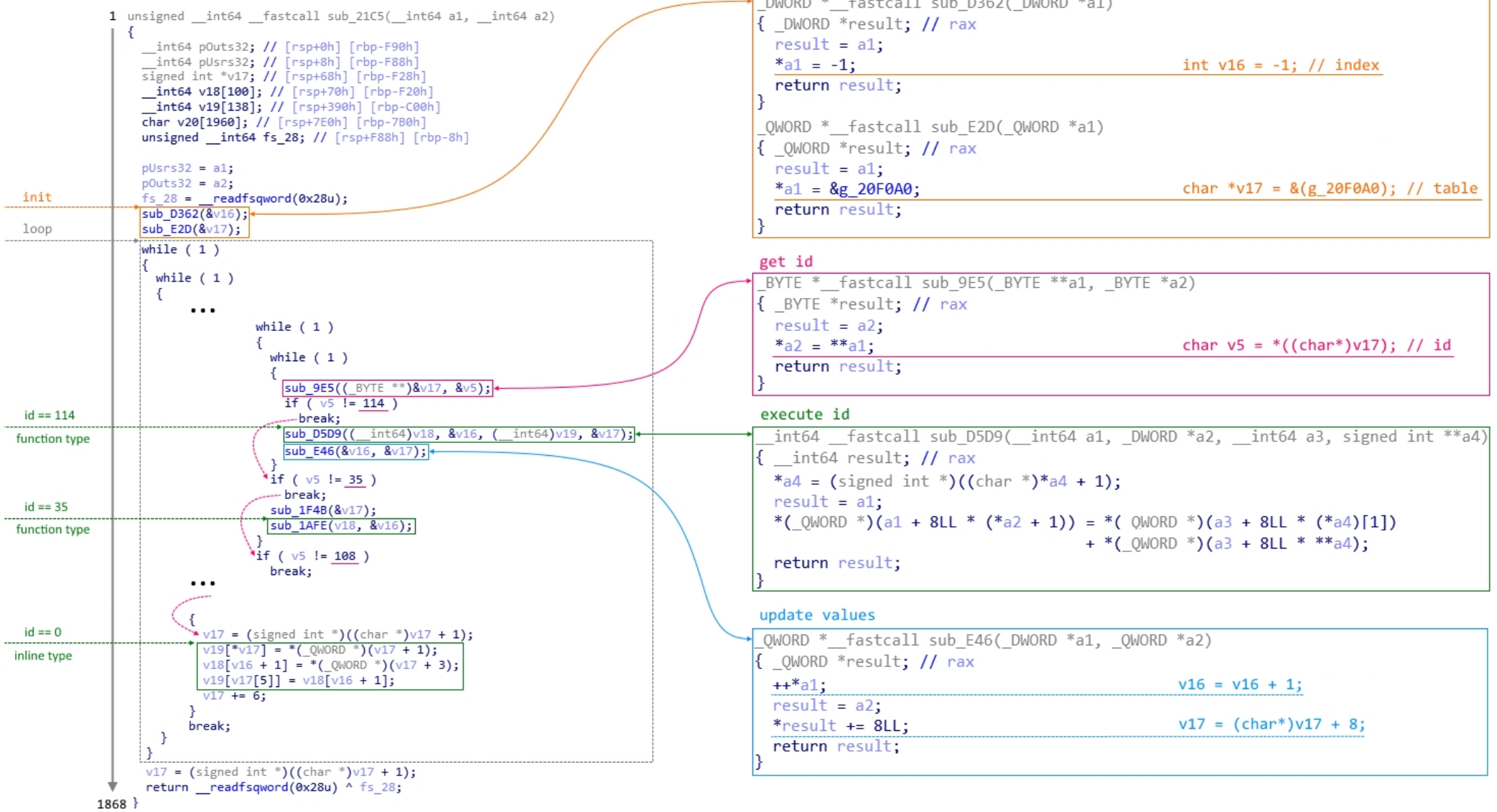
- v5 (id) block id
- v18, v19 (value) arrays of stored values
- v20 (point) array of control transfer points to blocks, of which only the entry point is used
- tmp (value) stored values accessed via pointers calculated relative to v20

Service variables include:

- v16 (index) index of the element from the array v18
- v17 (table) pointer to an element from the array g_20F0A0
- vF40, ... , vF80 (tmp) temporary elements

Stored **values** and elements from the bytecode array are **interpreted** differently **depending** on the **block** code.

sub_21C5() pseudo code



Pic. 4 Pseudo code of sub_21c5 subroutine

Step 2.3. The manual part of creating the code.

In the **types.h** file, we will define the necessary data **types**. The union type (**mem_t**) is used to access the same data through pointers of different types. The structure type (**obj_t**) is used to store all service variables and state variables. The remaining types relate to the native functionality of the code being created. In the **defs.h** file we will enter the necessary **aliases** that match the names of the variables from the pseudocode. In the **vars_g4c.h** file let's set the values of **global** variables.

We will create an **array of bytecode** using a hexadecimal editor (**Winhex**) (Pic. 5). In the **test.cpp** file let's create a **_t_test_()** function that repeats the pseudocode structure. Let's create the functions **sub_9E5(), sub_D362(), sub_E2D()** using their pseudocode from the decompiler (Pic. 7).

Step 2.4. The automatic part of code creation.

Python **scripts** use simple **text search** and substring **replacement**. To get the final result, they are run sequentially:

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

The **input** data is the generated **pseudocode** of the **sub_21C5()** procedure, pre-processed by manual replacement to **remove unnecessary** text.
The **output** is a **pseudocode** file (./tmp/t1_21c5.txt), in which pointer replacements are made to use indexes to access array elements.

Example:

```
*(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'
```

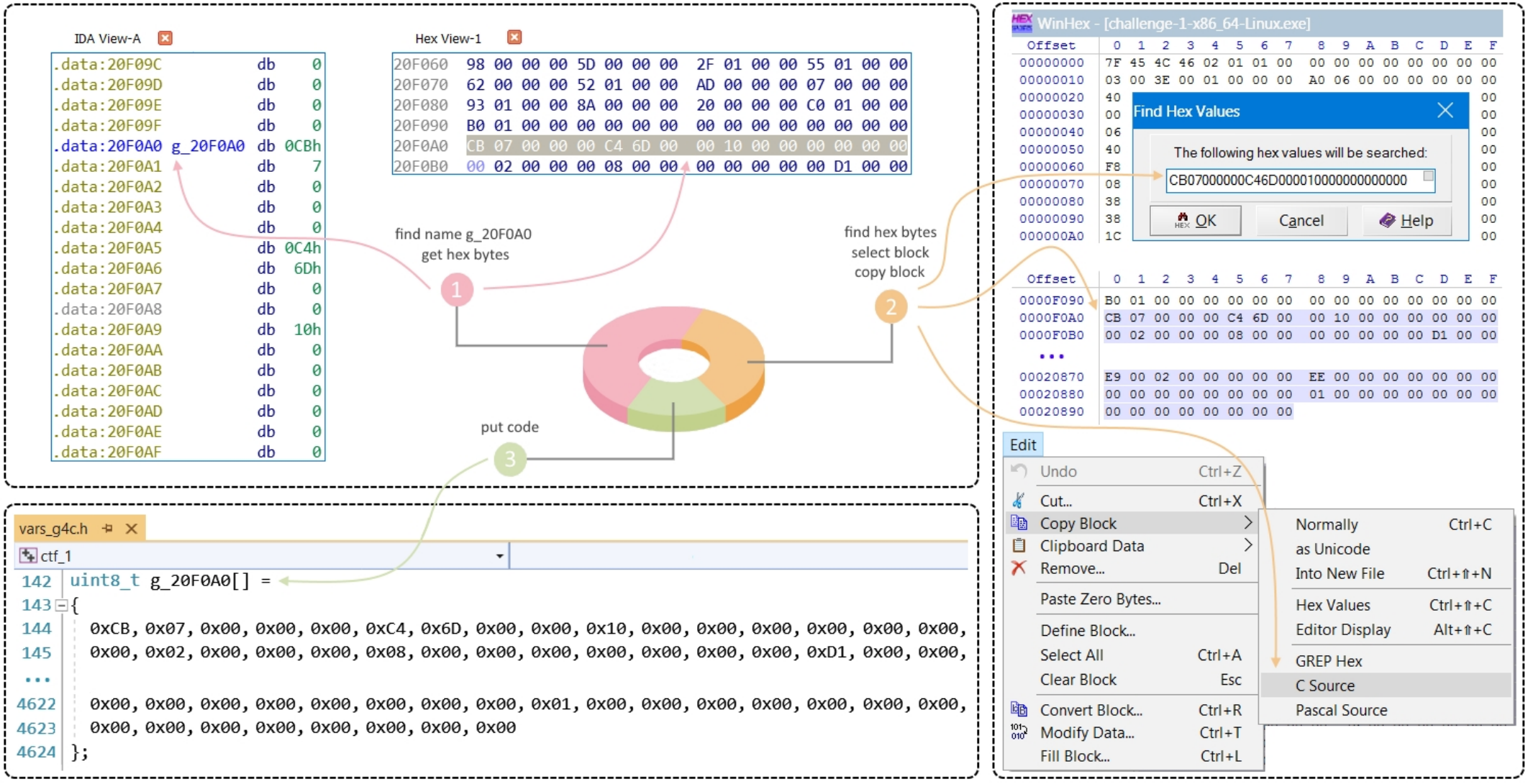
The **input** data is the **pseudocode** from clause 1.
The **output** is a **pseudocode file** (./tmp/t2_21c5.txt), in which substitutions of actions are made for calling the corresponding functions.

Example:

```
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_()

test.cpp

ctf_1

```
91
92  void _t_test_(obj_t *o)
93  {
94      _init_1st_(o);
95
96      bool brk = false, sts = false;
97
98      while (!brk)
99      {
100         _get_tag_(o);
101
102         tag_t _v5 = v5[0];
103
104         switch (_v5)
105         {
106             case (0x00): { t_00(o); break; }
107             case (0x01): { t_01(o); break; }
...              ...
220             case (0x87): { t_87(o); break; }
221             case (0x88): { t_88(o); brk = true; sts = true; break; }
222             case (0x8a): { t_8a(o); break; }
...              ...
318             case (0xff): { t_ff(o); break; }
319
320             default: { fprintf(vLog, "\ne: v5 := " F08_ " not found\n\n", _v5);
                           fflush (vLog); brk = true; break; }
321         }
322     }
323
324     if (sts) { _t_done_(o); }
325 }
```

```
_DWORD *__fastcall sub_D362(_DWORD *a1)
{ _DWORD *result;
  result = a1;
  *a1 = -1;
  return result;
}
```

```
void sub_D362(obj_t *o)
{   idx_t n = 0; n--;
    *(v16) = n;
    ...
}
```

```
_QWORD *__fastcall sub_E2D(_QWORD *a1)
{ _QWORD *result;
  result = a1;
  *a1 = &g_20F0A0;
  return result;
}
```

```
void sub_E2D(obj_t *o)
{   v17->tag = TAG_BASE;
    ...
}
```

```
_BYTE *__fastcall sub_9E5(_BYTE **a1, _BYTE *a2)
{ _BYTE *result;
  result = a2;
  *a2 = **a1;
  return result;
}
```

```
void sub_9E5(obj_t *o)
{   const tag_t *_v17 = v17->tag;
    {
        *(v5) = _v17[0]; // v17[0].tag[0]
        ...
    }
    o->mOrder++;
}
```

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'

The **input** data is the **pseudocode** from clause 2.
The **output** data are **c code files** (./output/t_<id>.txt), each of which implements the t_<id>() function for a block with an identifier (id).

Example for a file **t_0a.txt**:

```
void t_0a(obj_t *o)
{
    mv_tag_17(o, 1); // v17 += 1;
    {
        const idx_t *_v17 = v17->idx, _v16 = *(v16);
        ceq_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'

An **optional** service script that **checks** the file names from clause 3.

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

The **input** data is the generated **pseudocode for all functions** that are **called** from the **sub_21C5()** procedure, pre-processed by manual replacement to **remove unnecessary** text.
The **output** is a **pseudocode file** (./tmp/t1_fun_.txt), which is similar to paragraph 1.

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

The **input** data is the **pseudocode** from clause 5.
The **output** is a **pseudocode file** (./tmp/t2_21c5.txt), which is similar to paragraph 2.

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

The **input** data is the **pseudocode** from clause 6.
The **output** data is a **c code file** (./output/t3_fun_.txt), which implements the functions.

Example:

```
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);
}
```

Let's add all the created code to the solution. To perform its construction in the development environment, we will eliminate the errors that have occurred (**Pic. 8**).

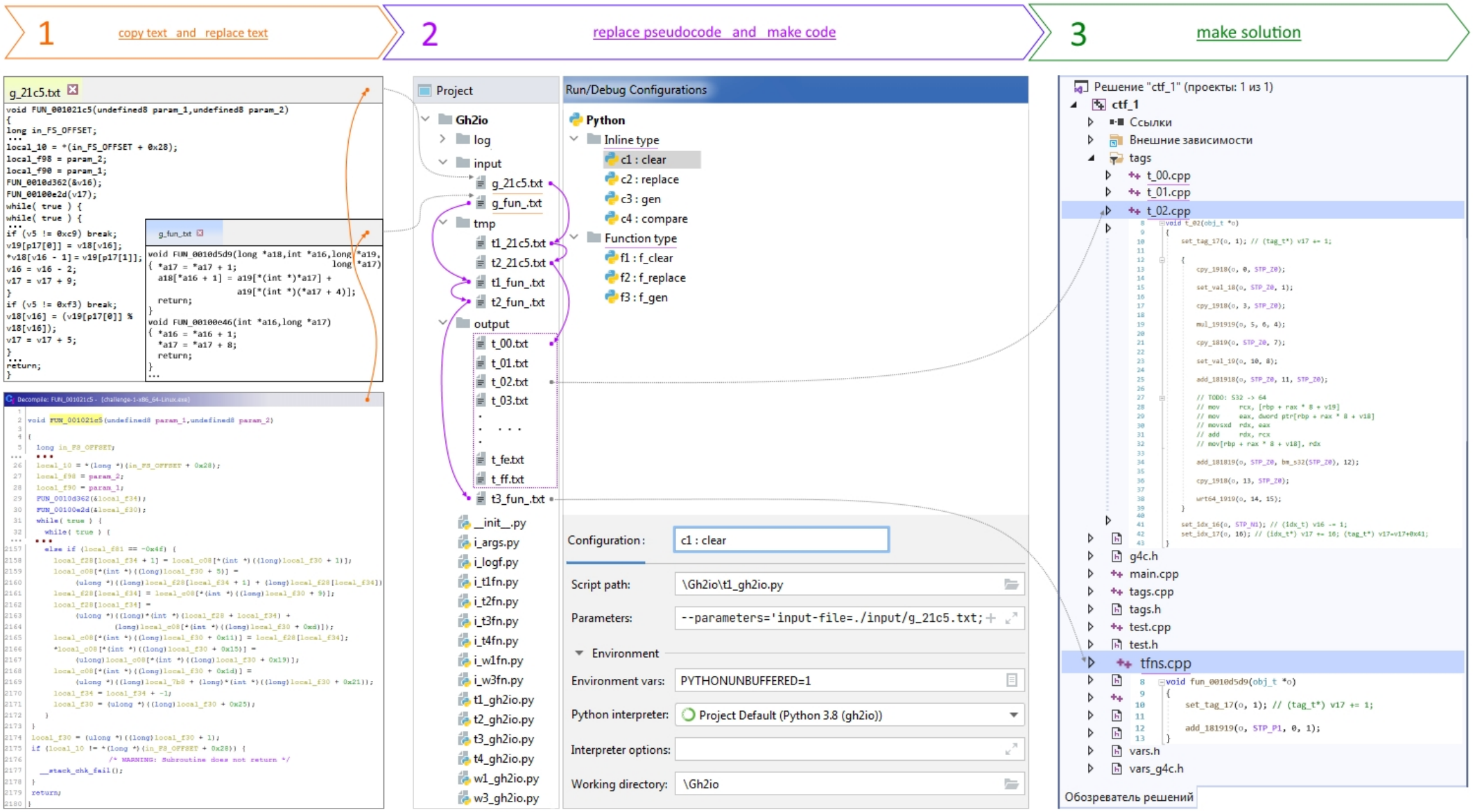
Remarks:
- preliminary manual replacement to remove unnecessary text is not implemented using scripts
- the variable size modifiers (8, 32, 64 bits) and their signed expansion are not taken into account, which leads to additional volumetric analysis
- macro definitions are not introduced for all numeric indexes, which excludes the use of their own data types to them

To implement the possibility of setting the **size of variables** and taking into account their **signed expansion**, we define **bitmasks** (**bm_t**) and functions for working with them.

Example:

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

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



Pic. 8 Make c code of ctf_1 solution

Let's set the default size of variables to 64 bits. By the characteristic **keywords**: **SLODWORD, LODWORD, LOBYTE** in the Idapro pseudocode we will find (substring **LO**) and fix the places that require the use of a **bit mask** (32-bit, 8-bit).

```
Example:

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));
```

Let's create customizable functions for testing and logging the execution of bytecode.

Step 3. Bytecode implementation.

It is required to create a c code implementing interpreted bytecode.

Let's set the **minimum** logging **settings** (small) and create a bytecode execution **protocol** by defining 2 user numbers.

Step 3.1. Analysis of the structure of the bytecode execution protocol.

The **protocol** consists of 2 parts: **initializing** (init) and **executing** (exec). In the **initializing** part, a **table** of control transfer **points** to blocks is filled in, of which **only** the **entry point** is **used**. The value of the **result** is **calculated** in the **executing** part. The protocol **token** (**s:**) means a change in the execution order of the bytecode, which corresponds to the transfer of control or **branching** in the source code. The beginning of the performing part follows a **unique** string:

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

The **length** of the performing part **varies** depending on the numbers entered.

Thus, to restore the **full** source **code** of the program, you will need to enter user **numbers**, ensuring the execution of **all** its **branches**.

Step 3.2. Analysis of protocol variables.

The **protocol** reflects the **values** and **addresses** of variables that contain input values, the **result**, working flags and arrays. The **initializing** part does **not affect the result**. Therefore, it may not be considered. Variable **addresses change** for each protocol.

The Python script searches for and replaces changing values.

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

The **input** data is the executing part of the **protocol** and the values of **addresses** and **variable names** specified in the script. Part of the addresses (**user, actn**) can be found at the beginning of the executing part. The other part (**calc**) can be found after performing the previous replacement.

Example:

```
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'))
```

The **output** is a **protocol file** (./output/log_am3.txt), which does not depend on the values of addresses and part of the values of variables (**Pic. 9**).

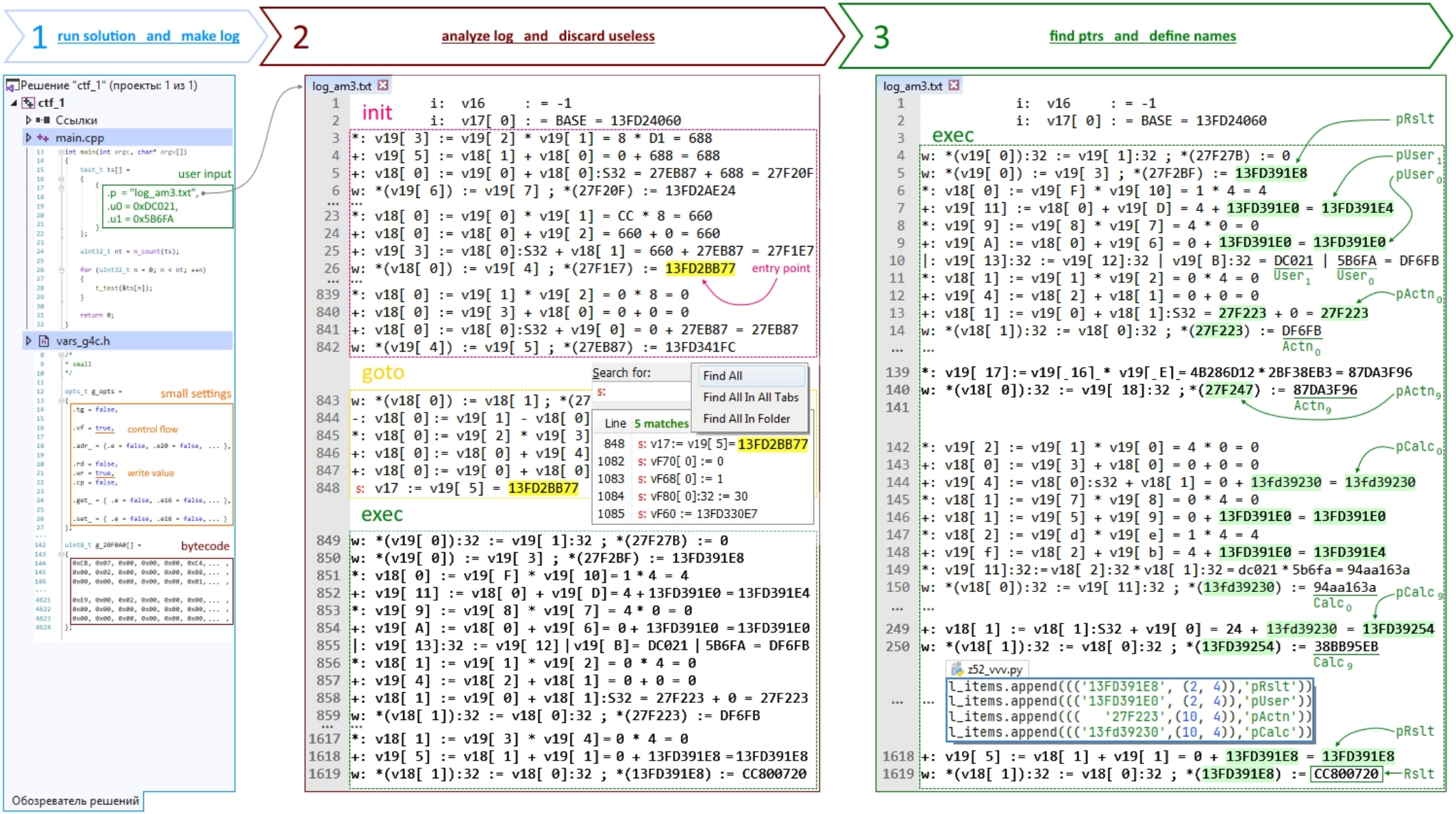
Step 3.3. Code creation.

Code creation consists of the following steps:

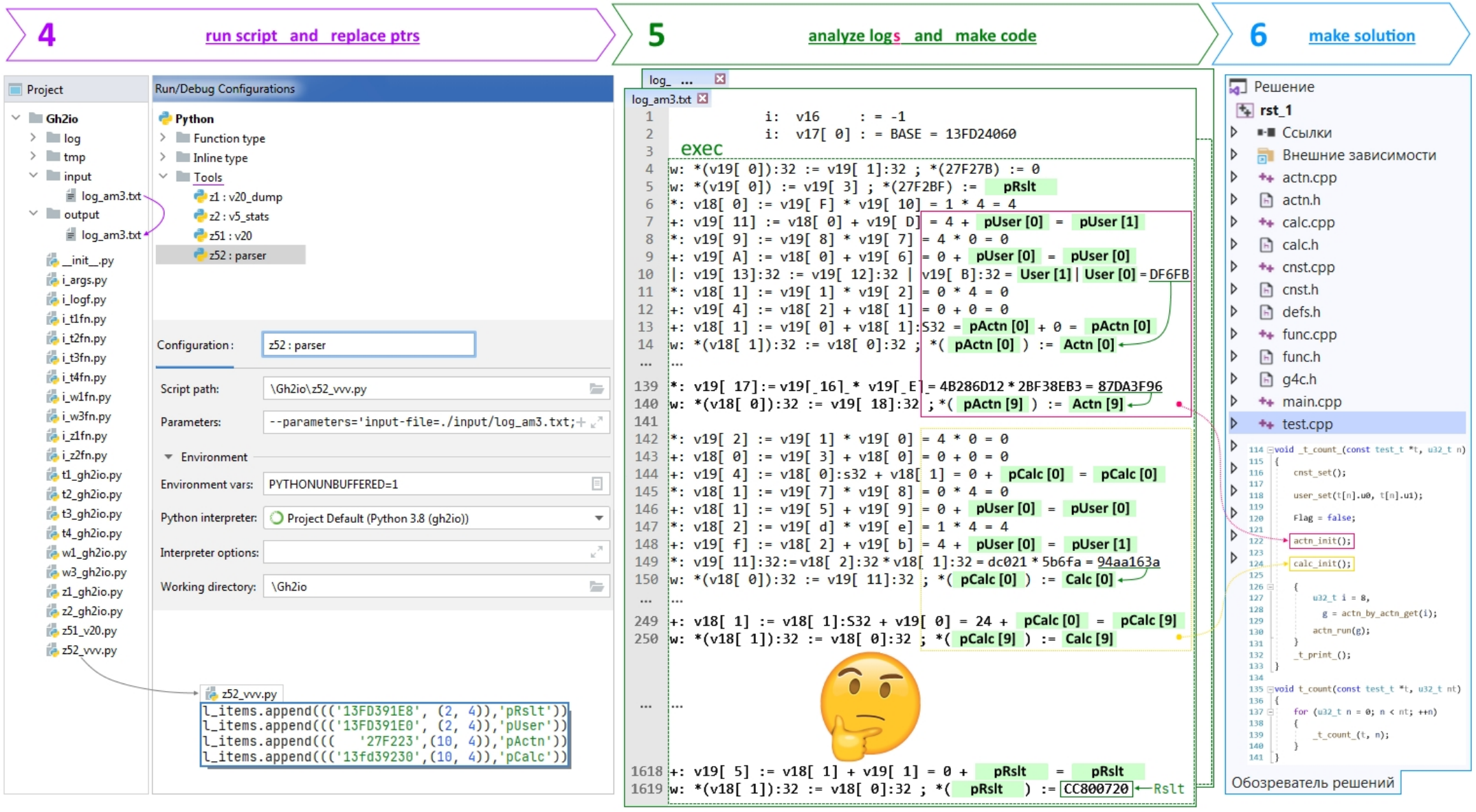
- we set 2 numbers
- creating and processing the protocol
- creating code that implements calculations from the protocol
- note all received code branches
- check the correctness of the result
- search for new numbers for one of the marked branches
- repeat all actions

The search for new numbers is performed by the scanner.

The scanner code is similar to the code being created, in which **accelerating** checks and condition interrupts are added (**Pic. 10**).



Pic. 9 Make c code of rst_1 solution: first part



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

Remarks:

- let's make a table of possible options using branch **coding**
- we will add options until we find the **reason** that causes the program to **crash**

Table of possible branching options:

Completed (custom numbers found):
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

Checked (no custom numbers were found, but code execution has already been implemented):
AM = 1, FM = 3
AM = 1, FM = 1
AM = 0, CM = 2, Flag = 0
AM = 0, CM = 1, CM1 = 1, SIX = ...

Never happens:
AM = 0, CM = 3, Flag = 1
AM = 0, CM = 2, Flag = 1

Not investigated:
AM = 0, CM = 1, CM1 = 2 ... 3

Part 2. Finding a solution.

Step 1. Analysis of the restored source code.

In the implementation of the **option**:

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

the **code** is present:

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

In this code, when a certain **condition** (**Flag**) occurs, the pointer for writing the result (**p_Rslt**) becomes equal to **nullptr**. An attempt to **write** the result using a **null pointer** will cause the program to **crash**. Therefore, this option is the desired **solution**.

Remarks:

- the condition is **composite**, which **complicates** the definition of user numbers
- **search** for custom numbers is performed by **brute force** using an **optimized** scanner, in which **slowing** cycles are **disabled**

Step 2. Search for custom numbers.

In the main() function, we will set the scan settings:

```
{ // AM = 1, FM = 0, G = 1, flag = 1 // access violation
    .log = {.p = "log_AM1_FM0_G1_flag1.txt" },
    .u0 = {.v = 0xffffffff, .e = 0x1 }, // scan range
    .u1 = {.v = 0xffffffff, .e = 0x1 },
    .ok = {.v = 1 }, // number of required solutions
    .am = AM_1, // branching parameters
    .fm = FM_0,
    .g = g_and_1
}
```

To **speed** up the search, the number of **required** solutions is **1**.

To **speed** up the search, we will set **interrupts** according to the conditions set in the settings.

In the function **actn_run():**

```
  {
      scan_t *s = Scanner;

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

In the function **func_run():**

```
  {
      scan_t *s = Scanner;

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

In the function **_fm0_loop_2_():**

```
  {
      scan_t *s = Scanner;

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

To **record** the found **solution** in the protocol, we will **set** the search **criteria**.

In the function **_fm0_g_and_1_():**

```
  { // speed up
      scan_t *s = Scanner;

      s->ok.b = (Actn[9] == 0xc4cfce5c); // <- criterion

      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;
  }
```

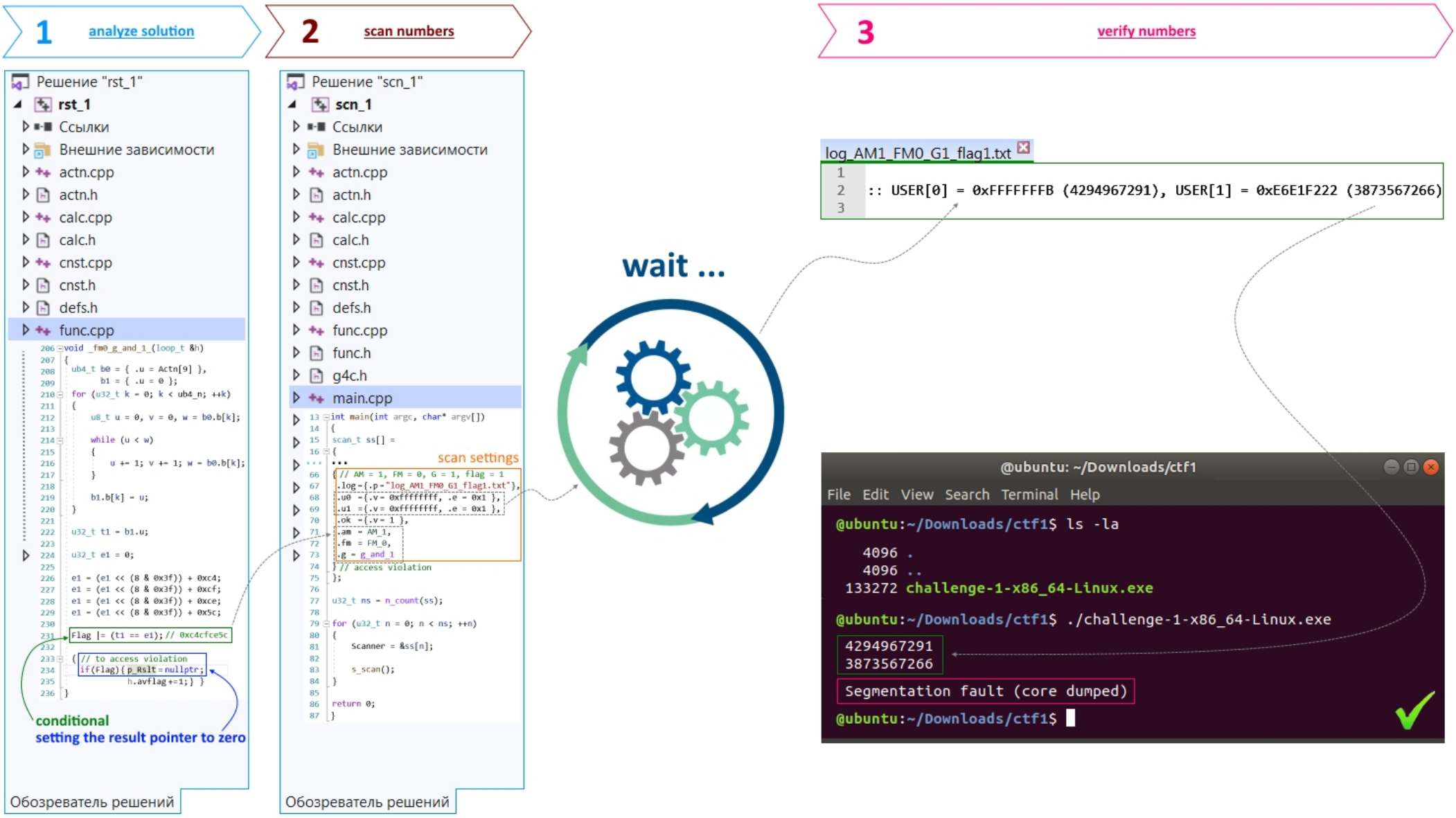
The protocol file (./log_AM1_FM0_G1_flag1.txt) contains the **user** numbers found.

Thus, the **desired** solution (**Pic. 11**):

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

Step 3. Conclusions.

Using the example of solving this task, a sequence of actions was created that allowed restoring the source code of the task, analyzing it and finding a solution. The sequence of actions is **partially automated**. However, informal **manual work** remains. To further **improve** the sequence of actions, it is required to **check** its operability on **other** tasks.



Pic. 11 Find an answer by scn_1 solution