

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.

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.

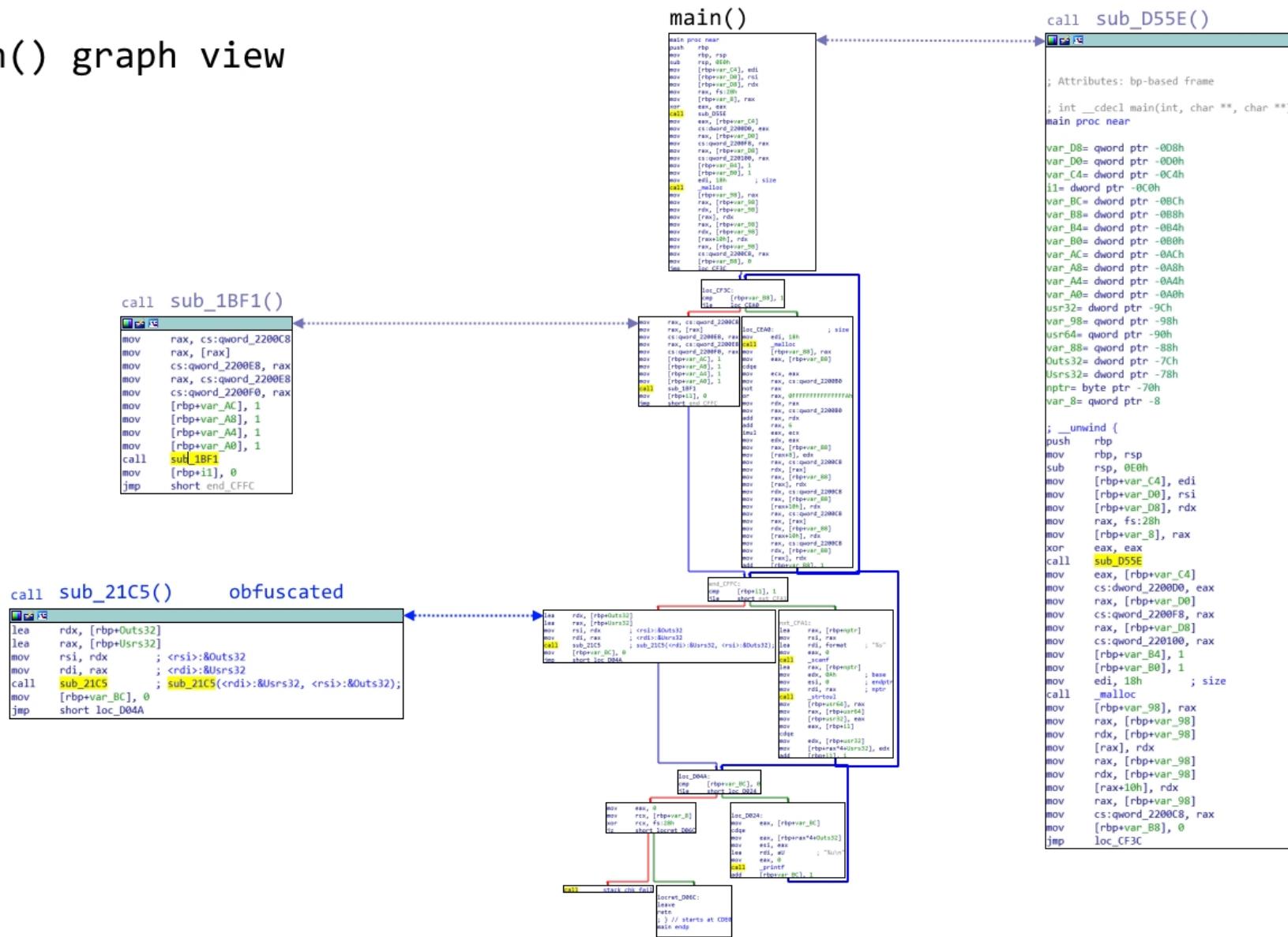


Round 2

- **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.

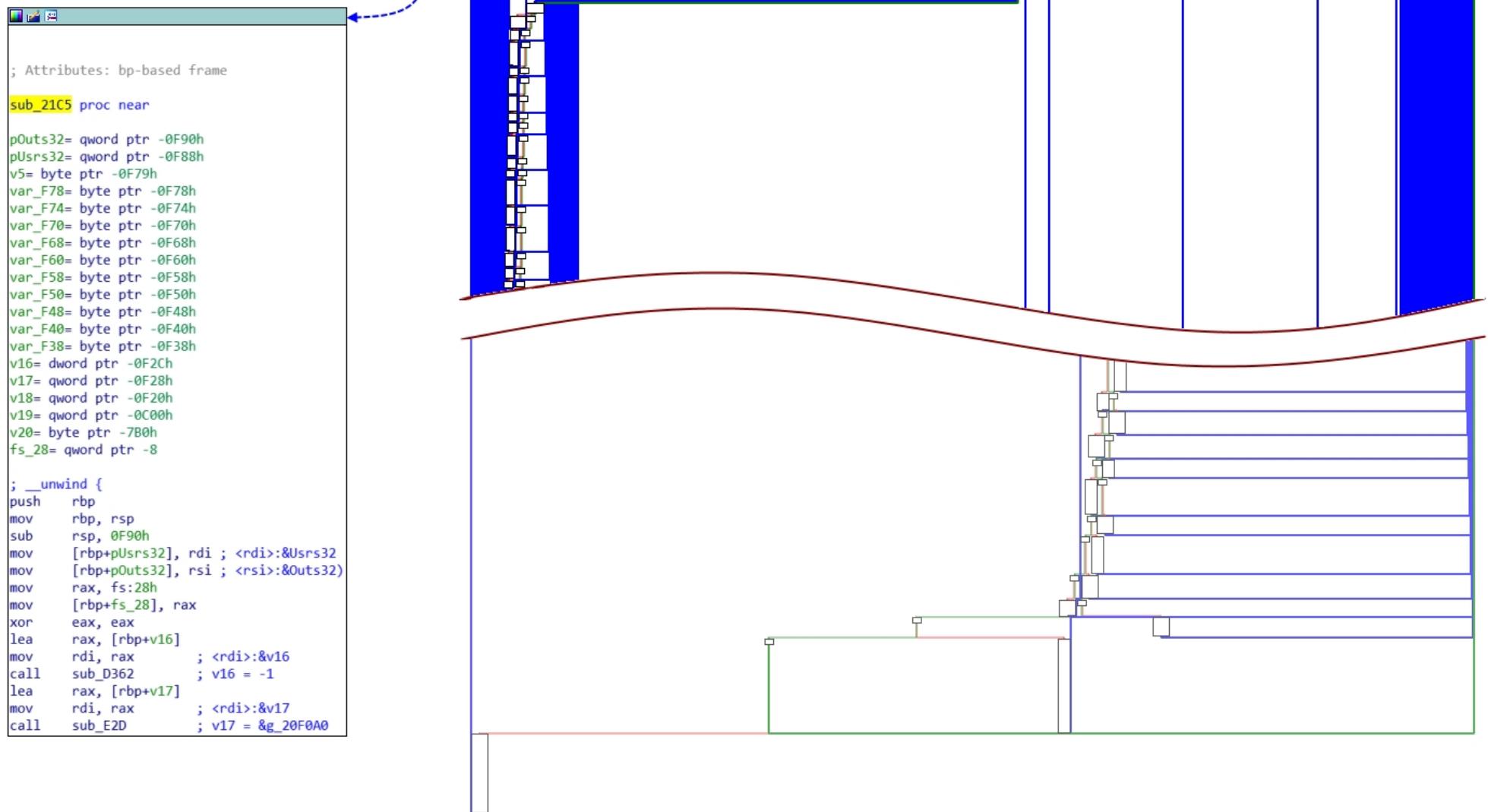
- —
- —
- —

main() graph LR



Pic. 2 Graph view of main function

sub_21C5() graph view



Pic. 3 Graph view of `sub_21c5` subroutine

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

```

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

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.

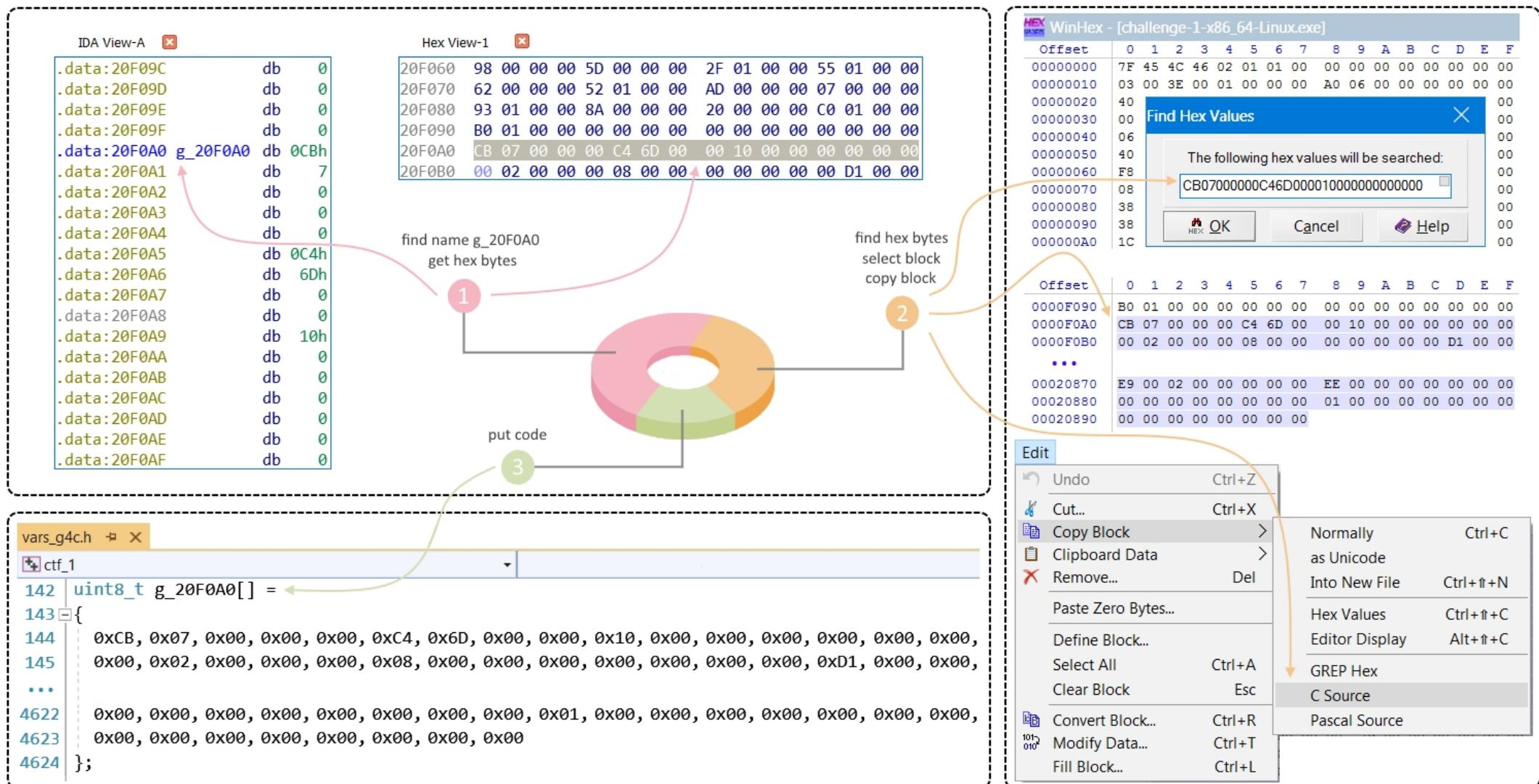
The **output** is a **pseudocode** file (`./tmp/t2_21c5.txt`), in which substitutions of actions are made for calling the corresponding functions.

Example:

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

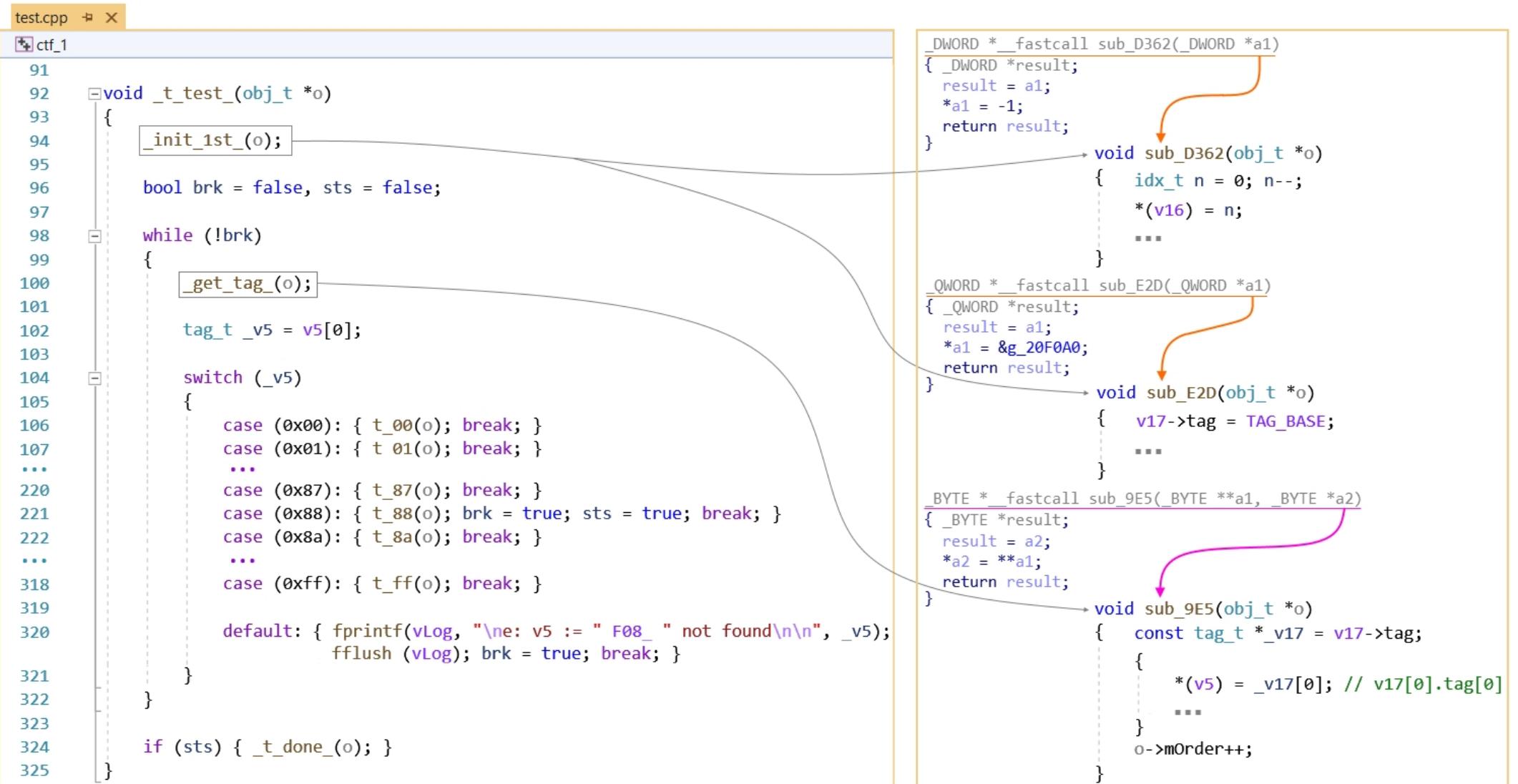
`v19[p17[1]] = (g_20F0A0 + p17[2]); -> ad_glb_19(o, 1, 2);`

make g_20F0A0[]



Pic. 5 Auto make c code of `g_20f0a0` byte array

make _t_test_()



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);
```

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    rrcx, [rbp + rax * 8 + v19]
    // mov    eax, word ptr [rbp + rax * 8 + v18]
    // movsd  rdx, eax
    // add    rdx, rax
    // 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

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).

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 showing assembly code with annotations. Annotations highlight memory addresses and variable names like `pRslt`, `pUser`, `pActn`, and `Actn`. A yellow thinking emoji is overlaid on the code area.
- Solution Explorer:** A sidebar titled "Решение" showing files for `rst_1`, including `actn.cpp`, `actn.h`, `calc.cpp`, `calc.h`, `cnst.cpp`, `cnst.h`, `defs.h`, `func.cpp`, `func.h`, `g4.h`, `main.cpp`, and `test.cpp`. A callout points from the `log_am3.txt` window to the `test.cpp` file.
- Code Editor:** A window showing the `test.cpp` file with handwritten annotations. Annotations point to specific lines of code, such as `actn_init()` and `calc_init()`.

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] = 0xFFFFFFFFB (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.

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     u32_t ns = n_count(ss);
28     for (u32_t n = 0; n < ns; ++n)
29     {
30         Scanner = &ss[n];
31         s_scan();
32     }
33     return 0;
34 }
```



3 verify numbers

log_AM1_FM0_G1_flag1.txt

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

```
@ubuntu: ~/Downloads/ctf1
File Edit View Search Terminal Help
@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