Static Program Analysis using Control Flow Graph and Symbolic Execution

For this lab session you will need the 64-bit Debian virtual machine provided by the instructor (user/password: `alumno`/`alumno`), as well as the source code files available on the workshop website (`https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/`). As prior knowledge, you need to remember how to use the Radare2 tool (command `r2`) or any other disassembler or debugger in Unix environments. At the end of this document you will find some links of interest, as well as a short summary of the Radare2 commands.

Specifically, in this session we will use the binary analysis environment `angr` (web page `http://angr.io/`). Reading [1] is recommended to learn more about this analysis environment. In short, `angr` is a modular environment developed in Python that allows binary parsing using VEX as an intermediate representation. In particular, `angr` allows you to perform the following types of analysis:

- Control flow graph:
- Data flow analysis;
- Difference between binaries;
- Disassemble;
- Symbolic execution;
- Automatic generation of exploits;
- and a long etcetera...

In this session, you will practice advanced binary static analysis techniques; specifically, based on the control flow graph and based on symbolic execution.

1 Based on Control Flow Graph

Binary analysis based on control flow graphs (*control-flow graph*, CFG) consists of building a graph with the assembly code of the binary. In this graph, the nodes are called *basic blocks* and the arcs between the nodes represent the relationship between the basic blocks. A basic block is a linear set of instructions that has a single input and a single output (that is, a basic block ends with an instruction that represents a flow change in execution).

Listing 1: File «ejemplo_cfg.c».

```
1  //gcc -o ejemplo_cfg ejemplo_cfg.c -no-pie
2  #include <stdio.h>
3
4  #define MAX 100
5  #define MIN 0
6
7  int read_valid_int(int min, int max)
8  {
9      int x = 0;
10
11     do{
12         printf("Provide a number x between %d and %d: ", min, max);
13         scanf("%d", &x);
14     }while(!(min <= x && x <= max));
15
16     return x;
17 }
18
19 int main(int argc, char* argv[])
20 {
21     int x = read_valid_int(MIN, MAX);
22
23     if (!(x % 2))
24         printf ("x is even\n");
25     else
26         printf ("x is odd\n");
27
28     return 0;
29 }
```

Look at the Listing 1. The program prompts the user for an integer between 0 and 100, which are constant values defined within the code. If the user does not enter a number in the mentioned interval, it is requested again, and so on until the number is within the given interval. Finally, in the main body of the program it is checked if the number is even or odd, informing the user accordingly. By default, `gcc` in its latest versions generates position-independent binary code (PIE). The CFG parser for `angr` can find problems with these types of binaries, so compiling without PIE is recommended.

Open a terminal in the virtual machine and compile the source code with the normal compile command in `gcc` and the `-no-pie` flag (i.e., `gcc -o ejemplo_cfg ejemplo_cfg.c -no-pie`). In the provided virtual machine you have the script `checksec.sh`, which shows you the software protections that a given Unix program incorporates. You can check your compiled program to verify that PIE is not activated.

1.1 Introduction to `angr`

We are now going to use `angr` to extract the flow control graph from the binary generated from Listing 1. There are two types of CFGs that can be generated with `angr`: a *fast CFG* and an *accurate CFG*. As the names suggest, generating a fast CFG is more efficient (in terms of time) than generating an accurate CFG. Although in general generating a fast CFG is enough, in this practice we are going to use the generation of accurate CFGs.

Each analysis of a binary will involve running a Python script to analyze it, telling `angr` what to do. First, you need to know that `angr` organizes binary analysis into **projects**. To initialize a new project, it's as easy as:

```
import angr

proj = angr.Project("./ejemplo_cfg", load_options={'auto_load_libs':False})
```

The `load_options` parameter in the code above is optional. In this case, it indicates to `angr` that it should not automatically load the possible dependencies that the binary under analysis has. This is because CFG analysis, on `angr`, does not distinguish between code coming from different binaries. This means that, by default, it will try to parse the control flow graph by iterating through the shared libraries as well. Note that this behavior is usually not the desired one, since the analysis time can increase considerably. With this option disabled, then, the flow is the one built from the given binary code, and not from other libraries.

Next, you have to tell `angr` which is going to be the root node of the graph. This is done using the following instructions:

```
main = proj.loader.main_object.get_symbol("main")
start_state = proj.factory.blank_state(addr=main.rebased_addr)
```

The first line takes care of looking up the "main" symbol in the binary. In principle, we are interested in generating a flow control graph from the function *main* of the binary (remember that in a C code it is the first function that runs). On the next line, the initial state is set to be the basic block starting at the address of the symbol *main*. If you are working with a binary whose source code is not available, you can set the address (in relative address) that you want to consider as the initial state. To specify the address, you can use the attribute `relative_addr`, `linked_addr` (compatible for binaries without PIE) or `rebased_addr` (compatible with binaries compiled with PIE). Note that `angr` provides a factory of objects to represent states.

Finally, to generate the flow control graph, use the function `CFGAccurate` (or `CFGFast`, if you are using the other type of CFG generator). This call has multiple configuration options, which may be of interest to a binary analyst during their work. Some of these parameters are detailed below:

- `context_sensitivity_level`: this parameter is used to indicate the level of context sensitivity. By default, it has a value of 1. It is recommended to read more information about this parameter and its implication in `https://docs.angr.io/built-in-analyses/cfg#context-sensitivity-level`.
- `starts`: a list of addresses to use as entry points in the analysis.
- `avoid_runs`: a list of addresses to ignore during scanning.
- `call_depth`: sets the limit of the depth of analysis to a certain number of calls. This parameter can be useful to check which functions are reached directly from a given function (by setting the value to 1).
- `initial_state`: initial state of the CFG.
- `keep_state`: to save memory, the state of each basic block is discarded by default. If this parameter is on, then the state is saved in the `CFGNode`.
- `enable_symbolic_back_traversal`: activates an intensive parsing technique to resolve indirect jumps.

- `enable_advanced_backward_slicing`: activates another intensive parsing technique to resolve indirect jumps.

In this example, we are going to make use of some of the parameters described above. To generate an accurate CFG, we insert the following Python code into the analysis script:

```
cfg = proj.analyses.CFGAccurate(fail_fast=True,
        starts=[main.rebased_addr], initial_state=start_state)
```

This line will generate an accurate CFG and store it in the variable `cfg`. Now, this variable can be queried to extract information of interest. Specifically, the CFG is a NetworkX digraph. That is, the functions of this library can be used with the variable that represents the obtained CFG. For instance:

```
print "This is the graph:", cfg.graph
print "It has %d nodes and %d edges"
               % (len(cfg.graph.nodes()), len(cfg.graph.edges()))
```

CFG nodes are instances of the class `CFGNode`. Depending on the context sensitivity that has been defined, it may be that a basic block is represented by multiple nodes in the graph (by multiple contexts). The following functions are illustrated as a working example with the CFG obtained (it is recommended that you test them to see their effects with the example being analyzed here):

```
# this grabs *any* node at a given location:
entry_node = cfg.get_any_node(main.rebased_addr)

# on the other hand, this grabs all of the nodes
print "There were %d contexts for the entry block" % len(cfg.
    get_all_nodes(main.rebased_addr))

# we can also look up predecessors and successors
print "Predecessors of the entry point:", entry_node.predecessors
print "Successors of the entry point:", entry_node.successors
print "Successors (and type of jump) of the entry point:", \
    [ jumpkind + " to " + str(node.addr) for node,jumpkind in cfg.
        get_successors_and_jumpkind(entry_node) ]
```

By default, `angr` does not have any function defined to be able to export the CFG to an image. There are some additional utilities (`angr-utils`, available at `https://github.com/axt/angr-utils`) that serve this purpose (and others). These utilities are already installed on the virtual machine provided by the instructor, so you can simply include them in your analysis script by adding these lines:

```
from angrutils import * # for plot_cfg

plot_cfg(cfg, "ejemplo_cfg", asminst=True, remove_imports=True,
    remove_path_terminator=True)
```

The analysis script that has been created will generate an execution control graph of the indicated binary, in image format and with the name (and path) of the file indicated in the call to `plot_cfg`. In this case, the CFG generated by `angr` from the compiled Listing 1 is the one shown in Figure 1.

Figure 1: CFG generated by `angr` from compiled Listing 1.

Finally, the analysis result also produces an object called *Function Manager*, accessible via `cfg.kb.functions` (where `cfg` is the parsing result variable). This object is accessed as a dictionary element in Python, and maps addresses to functions. Each of these functions also has various properties that may be of interest, such as the basic blocks that belong to a function, the string references of that specific function, etc. In `https://ntddk.github.io/angr-doc-ja/docs/analyses/cfg_accurate.html` (at the end) you can find more functions and their corresponding explanation[1]. For instance:

```
entry_func = cfg.kb.functions[b.rebased_addr]
# Set of basic blocks belonging to the function b, that you can explore
    and disassemble using capstone
list_bbls = entry_func.blocks
# list of all the constant strings that were referred to at any point in
    the function. They are formatted as (addr, string) tuples
strings_tuple = entry_func.string_references()
# Prints the name of the function
print entry_func.name
```

Execution of Analysis Script

`angr` works in a virtual Python environment, which allows you to separate required dependencies from different Python projects into different locations. Therefore, for the analysis script that has been prepared to work, we first have to indicate that we are working in the environment of `angr`:

```
workon angr
```

After executing this command, a textual prompt should appear on the command line to textually indicate you that you are in a different virtual environment.

`angr` Script Debugging

In the analysis script you can insert the instructions at any time:

```
import IPython; IPython.embed()
```

This line will provide the user with a Python shell where they can query the objects that have been generated, or perform other functions on the data. You can also use `pdb` to achieve the same result:

```
import pdb; pdb.set_trace()
```

Assignment 1.

Generate the script for precise CFG analysis as above and apply it to the binary generated from Listing 1. **Reason about each of the basic blocks that appear in the CFG, indicating with which parts of the Listing 1 they correspond.** Do not use any software protection for the compilation of the Listing 1.
You can also help yourself from other tools, such as Radare2, if you want to analyze the binary in more detail.

[1] If in the help of `angr` you see references to `b.entry`, in the new version it should be `b.*_addr` (*relative*, *linked*, or *rebased*).

2 Based on symbolic execution

In the previous part of this session you have better learned the concepts of control flow graphs, and applied your knowledge of *reading and understanding* of assembly code. Now, you are going to practice static analysis using symbolic execution.

Listing 2: File «ejemplo_symbolic.c».

```
#include <stdlib.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0

int e[] = {11, 17, 23};

struct data{
    unsigned int value;
    char *quote;
};

unsigned int offset = 0xBABEBEEF;
struct data food = {0xF00D, "To survive, we need\0"};
struct data coffee = {0x00C0FFEE, "In the morning, it wake us up\0"};
struct data dead = {0xDEAD, "No more sunlights seen, always we finish so\0"};
struct data *d[3] = {&food, &coffee, &dead};

int check_input(int value){
    if(value > 0 && value % 2 == 1)
        return TRUE;
    else if (value < 0 && value % 2 == 0)
        return TRUE;

    return FALSE;
}

int main(int argc, char *argv[]){
    char all_right = TRUE;

    if(argc != 4){
        printf("Usage: %s <int1> <int2> <int3>\n", argv[0]);
        return 1;
    }

    for(int i = 0; i < 3 && all_right; i++)
        all_right =  check_input(atoi(argv[i + 1])) &&
                (atoi(argv[i + 1])*((unsigned)e[i]) + offset) == d[i]->value;

    if(all_right)
        printf("WIN!\n");
    else
        printf("I'm really disappointed :(. Try next time\n");

    return 0;
}
```


Take a look at the Listing 2 provided. The program reads three input parameters, given by the user. Each of these elements is converted to an integer by calling `atoi`, multiplied by a fixed number, and another constant number is added as offset. Finally, it is checked if this result matches a certain value.

As before, we want the string `"WIN"` to be displayed on the screen to indicate that we have won. This string is displayed as long as the check condition is met with the three inputs provided. To achieve a match, the given input value must be controlled to ensure that the multiplication and addition operation overflows in such a way that the desired number is obtained, in addition to fulfilling a restriction that is imposed on each input depending on whether it is negative or positive.

How can we find the conditions that trigger the vulnerability and allow us to win? If we look at the space of possibilities that we have to cover, it turns out that each input variable is an integer, represented in C with 32 bits. Thus, we have 2^{32} possible input values for each of the possible values.

This search space is feasible today by brute force. In this exercise, then, it would be possible to use a binary analysis technique based on *fuzzing*. The *fuzzing* process could also be done in a more targeted way to save computation time, by imposing the constraints that the input be a particular set of characters (in this example, the characters `'0'` to `'9'`) .

However, we are going to learn how to fix it using symbolic execution. Remember that symbolic execution tries to exhaustively explore all possible execution paths of a binary. However, the binary is not executed, but is instead *interpreted* as follows:

- Abstract symbols are associated with program inputs, rather than concrete values.
- In each condition that produces a flow change in the execution, the state of the program is divided into two: one state is associated with the case in which the condition is fulfilled, saving the restriction associated with the variable involved in the condition; and the other state is associated with the case in which the flow change is not carried out, saving the negated restriction associated with the same variable. Then each of these states is evaluated separately.
- When the exploration of a path ends, it tries to resolve the associated restrictions until reaching that terminal state. Those paths that end in a contradiction are discarded as impossible paths, while for the others, concrete values are generated for some or all of the possible symbols. This last step is known as *concretization*.

We are now going to use `angr` to perform a symbolic execution analysis. First of all, compile the Listing 2 (compile it with the no PIE flag). As before, remember that `angr` works with projects, so a new project will have to be created:

```
import angr

proj = angr.Project("./ejemplo_symbolic", load_options={'auto_load_libs':False})
```

We now need to generate arguments to pass to the binary. Arguments can be concrete (e.g., Python strings, integers, reals, etc.) or symbolic. Since we don't feel like thinking much more, we're going to define the two arguments as symbolic variables. `angr` uses `claripy` as a constraint resolution engine, so we have to use its functions to define these symbolic variables:


```
import claripy

arg1 = claripy.BVS('sym_arg', 8 * 11) # maximum 11 * 8 bits
arg2 = claripy.BVS('sym_arg', 8 * 11) # maximum 11 * 8 bits
arg3 = claripy.BVS('sym_arg', 8 * 11) # maximum 11 * 8 bits
```

Note that the variables have been defined as an 11-byte symbolic bit vector (BVS). Remember that we are passing each of the arguments as a string and that the range of numbers needed by the program is a 32-bit number (maximum), which can be positive or negative; so the maximum length of the string we need to represent these numbers is 11 bytes ($-2^{31} = -2147483647$). Regardless of this prior knowledge that we have, for `claripy` they are symbolic arguments that can represent any binary data type.

Now, we have to define the initial state in which to start. `angr` uses SimuVEX to represent the state of the program, which provides a low-level, architecture-dependent representation of the binary being analyzed. Again, we make use of the `angr` object factory to generate our initial state:

```
st = proj.factory.entry_state(args=['./ejemplo_symbolic', arg1, arg2, arg3])
st.libc.max_strtol_len = 11 # tweak
```

With this, `st` will be our new state. Notice that the value of 11 has also been set to `st.libc.max_strtol_len`. This tells `angr` that the symbolic representation of the functions `atoi/strol` is at most 11 bytes (default is 10). Also notice that we are making the implicit assumption that our input parameter will be given to a `atoi` function. This usually requires prior analysis of the binary code using static analysis, or trial and error.

Now, let's explore the execution paths of our application. Specifically, we are going to tell `angr` that from the initial state we have defined, try to find a path such that the string `"WIN"` is written to standard output:

```
ex = proj.surveyors.Explorer(start=st,find=lambda p: "WIN" in p.state.
    posix.dumps(1));
```

Note that our state `st` has been defined as the initial state, and the verification of the path that fulfills what interests us has been indicated by means of a lambda function. The lambda function that has been defined can be read as *scans all paths looking for the path* `p` *such that in its current state* `p.state` *contains the string* `"WIN"` *in its standard output* (`p.state.posix.dumps(1)`, where 1 is the file descriptor for standard output). Here, a lot of concrete actions could now be performed on symbolic execution, such as:

- Run a step(`ex.step()`);
- Run up to 20 steps (`ex.run(n=20)`, to run up to 20 steps);
- Run all possible paths to the end (`ex.run()`;

This pathfinder can be given other instructions, such as telling it to:

- `find=0x804846b`, will try to find a path that leads to the address `0x804846b`;
- `find=lambda path : my_predicate(path)`, will try to find a path that satisfies the predicate indicated by `my_predicate` (this is the one we used here as an example);
- `find=addr1, avoid=addr2`,will try to find a path leading to address `addr1`, avoiding all paths leading to `addr2`.

You can find another example of using `angr` and symbolic resolution in `https://monosource.github.io/2016/06/solving-tut2-angr` (note, the commands indicated have changed a little bit). Here, we are going to tell it to execute all possible paths to the end:

```
ex.run();
```

Now, if we print the value of the variable `ex`, it should display something like:

```
<Explorer with paths: 1 active, 1 spilled, 1 deadended, 0 errored, 0
    unconstrained, 1 found, 0 avoided, 0 deviating, 0 looping, 0 lost>
```

This is indicating, in order of appearance, that there is an active path, a path that is stopped due to lack of resources (*spilled*), a path that has reached an invalid end, no path with error, no path whose address depends on a value given by the user, a path for which a solution has been found, and no path has been avoided, diverted, looped, or not considered. Of course, the active path could be further explored.

The variable `ex` contains various fields of interest, such as `ex.found` and `ex.deadended`, which are lists of paths. Thus, `ex.found[0]` is the first path found, and `ex.found[0].state` is its state.

We are going to take the path that has been found to analyze it in more detail:

```
found = ex.found[0]
```

Now, we can ask `angr` to specify the input values that allow us to get to this path. For example, considering the second argument:

```
found.se.eval(arg2, cast_to=str)
```

This command should return something similar to `'0069091343\xe0'`. Although we know that the input has to be a number, we have not restricted in our script the symbolic type of the parameters and therefore those characters "0" and "\xe0" appear, which are dispensable to generate our entrance.

Assignment 2.

Discover for yourself the specific values of the other two arguments and verify that they are indeed values that allow the victory string to be displayed.

Discovering other Aspects of Symbolic Execution

Let's see all the power that symbolic execution gives us using the code from the previous example. You have just seen that symbolic execution allows us to execute a binary **in its entirety**, exploring those paths that are most convenient for our analysis purposes.

However, with `angr` we can also symbolically execute *some parts* of the code, such as a particular function. To illustrate, let's try executing the `check_input` function from Listing 2.

Consider that now we want to execute this function symbolically because we are interested in knowing some value so that it returns true (or for which it returns false). Of course, this code that we are analyzing is very simple, but also think that our interest is to analyze binaries for which we do not have the source code (such as malware samples, to name a good example). That is, we would have to analyze the assembler of the binary to *understand* what causes the return of true or false in this function.

We are going to load this binary into `r2` in order to analyze this particular function by looking at its CFG. The CFG of a function can also be viewed from `r2` with the command `VV` (before you

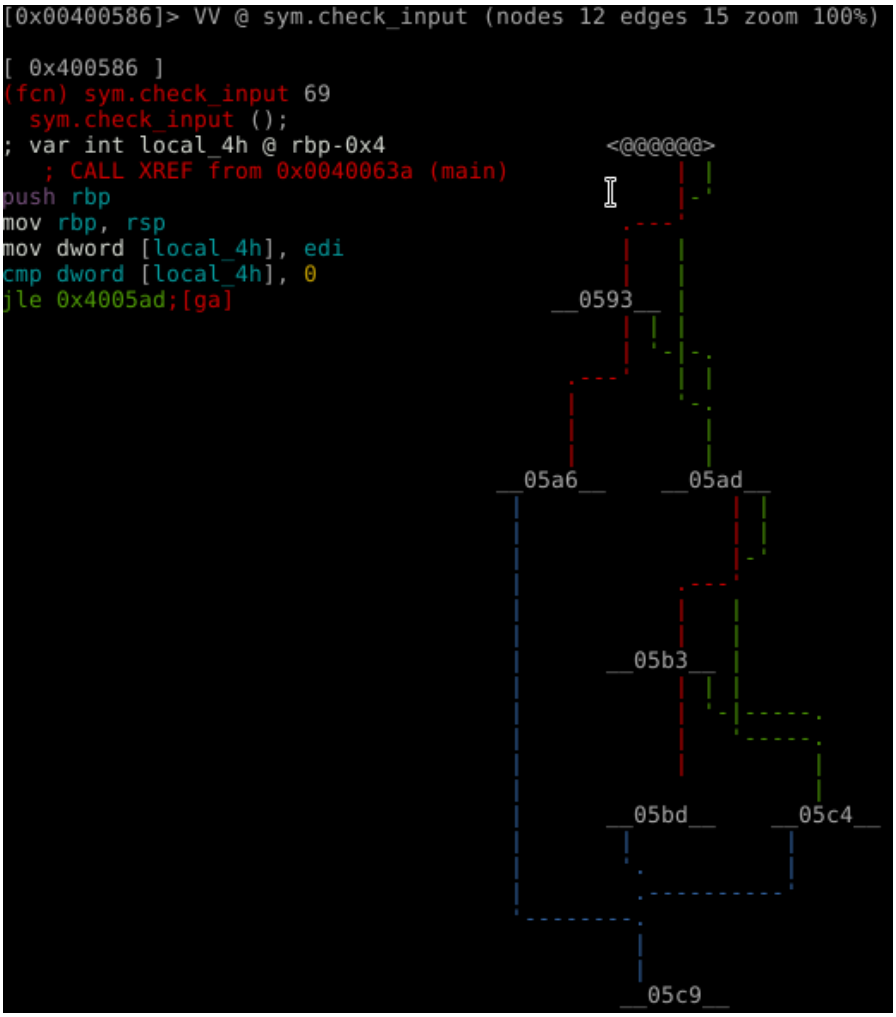

Figure 2: CFG of `check_input` function from the program «`ejemplo_symbolic`» according to `r2`.

will have to be located at the starting point of the function that interests you, that is, make sure that the console of `r2` displays the start address of that function of interest). After that, you can switch between views with the command `p`. Finally, on the left side of the screen you will see the basic block code corresponding to each node. You can navigate between nodes with the tab key. In this example, you will end up with something similar to what is shown in Figure 2.

Notice that the function is receiving the integer parameter `value` in the EDI register (basic block shown in Figure 2, address `0x400586`). Remember that in Intel x86-64 architectures the first parameters of the functions are passed through registers (whether they are integer parameters or pointers), in a certain order, and the following ones through the stack.

One of the basic blocks of this function, specifically the one at address `0x4005a6`, is the one that places the value of 1 (true value) in the EAX register (in the same way the one found at address tt 0x4005bd):

```
[ 0x4005a6 ]
mov eax, 1
jmp 0x4005c9;[gd]
```

These blocks correspond to the C code that is responsible for setting a value of true to the local variable that is returned later. Similarly, the basic block at address `0x4005c4` is the one that places a value of zero in the EAX register before continuing execution and reaching the last basic block, which is the epilogue of the function we are analyzing:

```
[ 0x4005c4 ]
mov eax, 0
[ 0x4005c9 ]
     ; JMP XREF from 0x004005c2 (sym.check_input)
     ; JMP XREF from 0x004005ab (sym.check_input)
pop rbp                                          __0586__
ret
```

We are now going to build a script so that `angr` executes this function symbolically. Specifically, we are going to start execution at the first basic block of the function (that is, address `0x400586`) and tell `angr` that the value of the EDI register is symbolic. We will then explore from that basic block all the paths that reach the final basic block (address `0x4005c9`) and avoid going through the block at address `0x4005c4`.

The `angr` script can be something like:

```
proj = angr.Project("./ejemplo_symbolic",
                load_options={'auto_load_libs':False})

check_input = 0x400586
state = proj.factory.entry_state(addr=check_input)

# function parameter comes at edi register
edi_start = state.se.BVS('edi_start', 8*4); # 32 bit register
state.regs.edi = edi_start

no_good = 0x4005c4
good = 0x4005c9
ex = proj.surveyors.Explorer(start=state, find=(good,),
                        avoid=(no_good,), enable_veritesting=True)
ex.run();
```

When executed, it returns us as a result:


```
<Explorer with paths: 0 active, 0 spilled, 0 deadended, 0 errored, 0
    unconstrained, 2 found, 1 avoided, 0 deviating, 0 looping, 0 lost>
```

Indicating that two possible valid paths have been found. If we analyze the value of the EDI register for each of them:

```
>>> ex.found[0].se.eval(edi_start,cast_to=int)
1
>>> ex.found[1].se.eval(edi_start,cast_to=int)
2147483648L
```

As you can see, one is a positive odd value, while the other is an even value. The value `2147483648L` is `0x80000000` in base 16, and remember that the first bit (which forms the value of 8 in the highest nibble of the number) is the one that indicates the negative value of the number (i.e., the value would be -2147483648) in two's complement. You can see it if it is expressed as a string instead of an integer:

```
>>> ex.found[0].se.eval(edi_start,cast_to=str)
'\x00\x00\x00\x01'
>>> ex.found[1].se.eval(edi_start,cast_to=str)
'\x80\x00\x00\x00'
```

Interestingly, despite having the rest of the bytes set to 0, the number is considered valid. If we analyze the function in more detail, the block that is checking this condition is:

```
[ 0x4005ad ]
cmp dword [local_4h], 0
jns 0x4005c4;[gf]
```

As you can see, it checks if the parameter is 0 by making a jump in case the sign flag is not activated, which explains the result obtained.

References

[1] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, pages 138–157. IEEE Computer Society, May 2016.

Links of interest

GDB debugger user manual `http://ldc.usb.ve/~figueira/cursos/ci3825/taller/material/gdb.html`

Basic GDB debugger commands `http://www.eis.uva.es/~fergay/III/enlaces/gdb.html`

radare2 Cheatsheet `https://github.com/radare/radare2/blob/master/doc/intro.md`

List of useful `r2` commands

- `r2 -d a.out` loads the program `a.out` for debugging

- `aa` analyzes the binary
- `afl` shows the list of functions identified in the current binary
- `pdf @ sym.main` shows the disassembly of the *sym.main* function
- `ad 40 @ esp` shows the 40 bytes, in double word format, from the address pointed to by the `esp` register
- `do` reloads the debugging session of the current program
- `db 0x0804480` places a breakpoint in the memory address `0x0804480`
- `db -0x0804800` removes the breakpoint from address `0x0804800`
- `dc` continues execution
- `ds` executes a statement and stops execution (returns the run control to the debugger; *step in*)
- `dso` executes a statement and stops execution (returns the run control to the debugger; *step over*) in the next statement
- `dm` shows the memory map
- `V` changes the view to visual mode
- `p` (in visual mode) allows to change between different views
- `F2` (in visual mode) sets a breakpoint
- `F7`/`F8` (in visual mode) executes a statement and stops execution (returns the run control to the debugger. `F7` corresponds to *step into*: in case of executing a `call` instruction, the debugger takes the control to the destination of the `call`. `F8` corresponds to *step over*: in case of executing a `call` statement, the debugger executes the calling code transparently and returns control right in the statement after the `call`
- `F9` (in visual mode) continues execution
- `?` evaluates an expression. For instance, `?v 0xFFFF + 1` would perform the sum of the two values, returning the result `0x10000`
- `/a ins` Look for the `ins` instruction (its encoding in hexadecimal) in the module where the r2 console is located

