System Security
Graded Assignment 2 - Reverse Engineering

Alessandro Cabodi

November 3, 2022

1 Exploring the binary

- What type of file is the provided executable? Run the `file` command on it. What is the meaning of *stripped*? How can you see whether this executable is stripped or not?

  Solution:

  By running the command `file` we see that it is an ELF 64-bit LSB PIE executable. Executable Linkable Format (ELF) is a common format for executable files, whilst PIE stands for Position Independent Executable, which means that the memory locations for the binary and all its dependencies are chosen randomly at every execution (this has some implications on the result we will see later when running `objdump -D` command).

  From the output of `file`, we may also notice that the executable is *stripped*. From the documentation, it means that symbolic information (and other information not required for execution) are removed from the executable file. Indeed, whilst with normal compilation the executable carries some debugging information (for instance the names of the variable or which line of the source code generated a certain instruction), such information has been removed in a stripped file, in order to reduce its size and save space. As a further confirmation, we could also try and run the command `objdump -syms` and verify that the symbol table is indeed empty.

- Often the easiest way to get information about the behavior of a binary is to run it! Execute the binary and play around with it, for example by entering your Legi number.

  Try to `strace` the program while you run it. What does `strace` report? Can you spot something interesting?

  Solution:

  From the documentation, `strace` command allows to capture, monitor, and troubleshoot programs in a system. In particular, it shows the system calls made by the program under analysis. It can be useful to understand why a program is not working as expected or what other files the program depends on (by looking at `open` system calls for instance).

In particular, by running the command on our executable and inspecting the output we find the following line: `openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3`.

System call `openat` is essentially the same as `open`, whilst `libc.so.6` should be a symbolic link to the location of the `glibc` library, which refers to the *GNU C Library*, the most widely used C library on Linux. Therefore, we have now discovered that the executable in question originates from a C program.

Other than that, we find the expected `read` and `write` system calls, to read the legi number, our requests and to print strings and results. In particular we can notice that when a request is issued a `clone` system call is invoked to open the expected browser page (due to frequent timeouts when the program attempts to open Firefox, it is difficult to proceeds with further analysis).

- The first step to reverse engineering an executable is interpreting it as assembly code – its (more or less) human-readable form. Use `objdump -D` for this. Understand the output, which shows you the binary itself on the left and its interpretation as operation *mnemonics* and their operands on the right. What do all the function names you can see have in common? (Remember your answer to the first question.) Find the main the `main` function. What is its address?

Solution:

We run the command `objdump -D` command on our executable and inspect the output.

Different sections are disassembled and shown. Scrolling down we reach the `.text` section, the one containing the code. We see that no functions can be distinguished in this section, as it would happen were the executable not stripped: the disassembled section is displayed as a unique block of operation mnemonics. In particular, notice that if the executable were not stripped we would be able to run command `gdb` and then `disassemble main`, but this is not possible since no symbol table is loaded, being the executable stripped.

We can also notice something interesting by looking at the address of such instructions: being this a PIE executable, only relative addresses are shown (that is, we know the offset, not the absolute location in memory).

To find `main` we then need to figure out a way around. Recall that from strace command we already know that glibc library is being used, therefore it is easier to make some assumptions.

In particular, exploiting GNU debugger (by running command `gdb`) and then `info files` we discover that the entry point is located at (relative) address 1420. Either by using gdb command `x/100i 0x0000000000001360` or by inspecting `objdump -D` output (the result is almost identical) we can than look for the first function call after the entry point. Thanks to previous knowledge, we may assume this to be a call to `__libc_start_main` function. Such a function takes as first parameter the actual main function. The first input parameter of a function is the last to be

prepared in assembly, hence we can look at the previous line with respect to the call to find that the parameter being prepared is at address 0x1360.

In conclusion the (relative) address of the `main` function is indeed 0x1360 (incidentally it is also the first line in text section, but this is not to be taken as a rule). An easier approach would be to use a disassembler like ghidra, which also leverages a decompiler (among other features): it is able to show an actual name for `__libc_start_main` thus speedily validating our previous assumptions.

2 Understanding the Student Edition

- To find the function that prints the very first line, find the string in the strings panel and find the function it is referenced from. We refer to this function as `function0` in the following. Which function is this? What is its address as reported by Ida/Ghidra?

  Solution:

  By inspecting the string panel in Ghidra looking for the first line printed when executing the program, we find that such string is associated with data `DAT_00102977` (which we can hereby renominate). This is taken as parameter by `puts` inside the `main` function. We can conclude that `function0` is `main`, with address 00101360 (which corresponds to what we found out manually before; notice that the first part of the address is only an assumption made by Ghidra).

  Study carefully the initial screen that Ghidra or Ida give you: there is assembly code divided in *basic blocks* and forming a *control flow graph*. What is a basic block in this context? How are basic blocks usually separated? What is a control flow graph?

  Solution:

  A control-flow graph (CFG) is a graph representation, whose nodes are referred to as *basic blocks*, of all the possible paths that might be traversed during the execution of a program.
  A basic block in this context is a set of assembly instructions. A jump target starts a new basic block and jumps from a block end that block.

  By clicking on *Function Graph* in Ghidra we have several views at our disposal. As reported in the documentation, this command displays the flow chart of the current function. The colored egdes of the flow chart represent the outcome of conditional jump instructions. Green means that the condition is satisfied, red means not satisfied.

- `function0` does not seem to be the one that asks for your Legi – but you can spot a function call early on in the function body. We'll refer to the called function as `function1`. What is the name of `function1` according to the decompiler? Does the name represent the purpose of the function – why / why not?

  Solution:

Indeed, `main` does not directly ask to enter out ID, but early on it calls `FUN_00101980` that does. A proper function name is not displayed: the address is used as function name instead. Again, since the executable is stripped the symbol table is empty, hence no function name can be retrieved by the disassembler. Therefore, it is convenient to rename such function accordingly.

- Now, find out what `function1` does. Remember to rename variables and function parameters when you have a hypothesis on what their purpose is to keep track of your progress. Similarly, when you think you figured out what the function does, rename it with a tentative name. What could be a good tentative name for `function1`? What does `function1` do?

  Solution:

  `function1`, that is aforementioned `FUN_00101980`, is the one in charge of reading the user legi ID. It only performs simple checks on the correctness of the format (specifically the field XX should be lower than 100, and the other two XXX should be lower than 1000), the it returns the legi ID as (unsigned) integer (e.g. 12345678). Therefore, a tentative name could be `read_legi()`.

  Notice also that returned ID is taken as input by the subsequent function in `main`.

- Go back to `function0`. There, you can see the value returned by `function1` gets passed to another function. We'll call that one `function2`. Before reverse engineering `function2`, first understand how it is embedded in `function0`. This will help you later when you do a deep dive in it. Which possible return values can we spot from the checks that `function0` performs with the return value? What does each value correspond to in terms of access level? To answer this, think about what you already know: the string that the program prints when you enter your own Legi, the strings you can see in the binary, and what the program does with the return value.

  Solution:

  `function2` is `FUN_00101509` immediately after `read_legi`, whose return value is passed to `function2`, which we hereby call `access_control`. Indeed, inspecting the `main` body, we can see that the value returned by `int access_control(legi)` is used to activate either the lecturer edition or the student edition. We may call the return value `access_privilege` instead of default `iVar` for clarity's sake.

  In particular if `access_privilege == 1`, by cross checking the `DAT` parameter with its associated string, we see that that the Student Edition is activated. A function call follows.

  Instead, if `access_privilege == 2`, by cross checking the `DAT` parameter with its associated string, we see that that the Lecturer Edition is activated. Two function calls follows such activation (we will learn we will be asked to provide a product key).

Finally, if `access_privilege != 1 && access_privilege != 2` the check fails and the program outputs string "This program was registered to a different user" (by looking at primary label `LAB_00101403`).

- Now you should have a good understanding of which value `function2` should return when passed (i) your Legi; (ii) some other specific number that you still do not know; and (iii) any other value. Let's dive deep into `function2`. Read the code and rename variables according to your understanding. Use your knowledge of the function return values as well. What is the mathematical operation that is done on the input Legi value? How does the program hide the expected Legi values, to avoid being detected by, e.g., simple string search on the binary?

  Solution:

Thanks to previous analysis, we may assume that if my student legi is passed as parameter, `access_privilege` variable will be set to 1, and the Student Edition will be activated.

If some specific number, that is a lectured ID, that we don't know yet is passed, `access_privilege` variable will be set to 2 instead, and the Lecturer Edition will be activated.

For any other value of `access_privilege` variable the check will fail.

Let's now dive deep into `access_control`. Trying to refactor the code, this is what the result looks like:

```
uint access_control(uint legi) {

  uint user_access;
  int lecturer_access;
  ulong validUser;

  validUser = ((ulong)legi * 61658338) % 100000007;
  user_access = (uint)validUser;
  if (validUser == 1) {
    lecturer_access = 0;
    user_access = 1;
  }
  else {
    validUser = ((ulong)legi * 8691641) % 100000007;
    lecturer_access = (int)validUser;
    if (validUser == 1) {
      user_access = 0;
      lecturer_access = 1;
    }
  }
  return lecturer_access * 2 | user_access;
}
```

We see that mod operation is performed on legi ID number (times a seemingly random scalar factor). In this way, subsequent equality check on `validUser` in `if` condition takes simply 1 as second argument (`if (validUser == 1)`), not the expected student or lecturer ID (that is we don't have a condition like `if (ID == validID)`).

Therefore, a simple string search on the binary does allow to retrieve a valid student or, most importantly, lecturer ID.

- Go back to `function0` and reverse the function (referred to as `function3`) that gets called when execution takes the `if` branch corresponding to inputting your Legi number. What does `function3` do? What are its most frequent types of operations? How does the prompt work – which commands does it accept and what do they do?

  Note: At the end, it seems to call a function pointer, the value of which is not known until runtime – ignore it for the moment.

  Solution:

  By inputting my legi number, the Student Edition activation branch is executed. A function call is performed. The body of such function is empty except for a further function call (we can sort of ignore this aspect and consider this last function as being directly invoked by `main`, that is we may consider this to be the effective `function3`), which takes as input a pointer to function (which we ignore for the moment, as written in the note).

  Let's dive into `function3` body.

  Early in the function body, immediately following the "image" displayed at the very beginning of the program execution, we find a primary label devoted to reading user input (basically it consists of a do-while loop where input is checked not to be empty). After we enter do-while loop if end of file is reached, an `exit` is issued (we may call this label `exit_branch`).

  The function then performs some operations based on the given input. We see here the most typical operations: if either "nothing" or "q" or "exit" is given in input, the program executes the aforementioned `exit_branch`.

  Otherwise, a call to the function pointer received as parameter by `function3` is executed. At this point we may assume such function pointer to be devoted to solve the request input by the user (we then call such function pointer `execute_request`: it takes as parameter the request input by the user). Based on the return of `execute_request`, the program reports either a failure or a success by printing in standard output the appropriate string. A `goto read_input` is then issued, and the whole process explained above is repeated.

- With the above function, we now reversed all functions related to the Student Edition. We need to go back to the only finding we did not use yet: there seems to be a Legi input for which the program unlocks the Lecturer Edition. What is this value? How did you compute it?

Solution:

Recall our previous analysis on `access_control` function. The Lecturer Edition is activated if $(legi \cdot 8691641) \bmod 100000007 == 1$.

Hence it is sufficient to write a very simple python script like the following:

```
for id in range(10000000, 99999999):
    if id * 8691641 % 100000007 == 1:
        print(n)

=> id = 31337042
```

We scan through every possible valid input ID, until we find one that satisfy the condition present in `access_control` function as explained before. As a result, we get $ID = 31337042$ which is the valid lecturer ID. By inputting 31-337-042 when asked to enter our ID we are then able to finally activate the Lecturer Edition.

Notice however that at this point we are now asked to provide a valid product key, which we don't know yet.

3 Unlocking the Lecturer Edition

- First, get some fast-track satisfaction and immediately access the Lecturer functionality by patching away this check: you can use Ida, Ghidra, or a *hexeditor* to edit the executable and create a copy that does not have the check. To do this, you can adopt different approaches, e.g., `nop` away the function call, or change the `jmp` or `cmp` of some parts of the code. What approach did you choose? Describe your solution in detail. Check that it works by running your modified executable and leveraging the raw power of our AI.

Solution:

By inspecting the Lecturer Edition Activation `if` branch, we immidiately see two function calls, which are hereby called, respectively, `check_productKey` (which takes the legi ID in input) and `lecturer_edition_start`.

As the name suggests, we indeed discover that the first function ask for the product key early in its body. The rest of the function is devoted to checks on the input key. While it would be possible to appropriately change `JMP` or `CMP` in some parts of the code, a simpler approach is to simply `NOP` away such function call, as it does not influence in any way the rest of the program execution.

This is the original disassembled code section (a part from some renaming) where `check_productKey` and `lecturer_edition_start` functions are called:

```
001013e2 89 ef          MOV       EDI,EBP
001013e4 e8 07 09       CALL      check_productKey
         00 00
001013e9 31 c0          XOR       access_privilege, access_privilege
```

```
001013eb e8 95 07      CALL      lecturer_edition_start
        00 00
```

Let's analyse such instructions line by line.

The first `MOV` instruction is to move content of register `EBP` in register `EDI`, which is then taken as parameter by `check_productKey`, which takes the legi as input. Indeed, scrolling up a few lines we see that legi was previously placed in that register. Since legi is not accessed anymore in the rest of the code, we need to remove such instruction (that is, we need to substitute it with `NOP`), otherwise it will be taken as parameter by `lecturer_edition_start`, thus compromising its signature. The next line contains the call to `check_productKey`, hence it must be obviously removed. `XOR` instruction in the next line should not be removed as it serves to reset to 0 the register where access privilege value is stored.

This is how the disassembled code section looks like after the appropriate modifications:

```
001013e2 66 90         NOP
001013e4 90            NOP
001013e5 66 90         NOP
001013e7 66 90         NOP
001013e9 31 c0         XOR       access_privilege,access_privilege
001013eb e8 95 07      CALL      lecturer_edition_start
        00 00
```

To change instructions in Ghidra, we need to click on patch instruction. When `NOP`ing away instructions we are given the choice to cover either 1 or 2 bytes (respectively with `NOP 90` or `NOP 66 90`), which allow for more compactness. The final effect is that the function call to `check_productKey` completely disappears.

Hence, immediately after checking the lecturer ID, function `lecturer_edition_start` is invoked and we now have full access to the Lecturer Edition.

- In software piracy, what you did above is called a `crack`: a modified executable that bypasses license checks (here, the license number) for unauthorized software use. Another approach is instead to understand how the license check works and writing a generator of valid inputs that bypass the checks—a *keygen*[1]. Let's start to understand how the license code check works. We can observe that the first thing it does, to read one's license input, is to convert it into a different format. Which input format does the license code have? How does the program read and interpret this value? Where is this conversion happening?

  Solution:

[1]This type of illicit software is being phased out with, e.g., online activation checks.

The function records user's input by calling `fgets` function early on in `check_productKey` function body (inside a do-while loop). Therefore, input is initially stored in a string, that is variable `input` is of type `char *`. Being this a 64 bit application, the size of a char pointer is 8 bytes (as also suggested by Ghidra).

After some preliminary check (basically to verify input is not empty), we enter a further while loop where the actual check on the correctness of the product key is performed. At every iteration of the while loop we take each character from the inserted product key and some validation checks are performed.
First, we notice that the key has to be 64 digits long. Since every digit is a character, the product key string needs to be stored in 64 bytes.
A sequence of `if` statements follows: we may infer that every digit will be interpreted as a hexadecimal since the function only accepts either numerical characters or letters between 'a' and 'f' (also in uppercase). Next, we may notice, also by looking at the register sizes, that the function, through some bitwise operations, stores each digit on the stack in 4 bits (here some debugging may be of help, as the decompiled operations are a bit convoluted).

In the end, when the loop terminates, the stack will be populated with the product key digits, each of them interpreted as a hexadecimal value and of size 4 bits (that is, half the initial size). As a consequence the product key is now stored in 32 bytes. As we will see, in the following, most checks on the product key are performed byte by byte, that is, taking two product key digits at a time.

Let's dive deep into the function that seems to perform the license check. What does it do and how does it operate on the user input? How many "helper" functions does it use for its checks? Describe each in detail.

Solution:

`licence_check` function takes as input the product key (loaded on the stack by the parent function) and the legi number. At the moment, the product key only needs to be input in a correct format, as we must still discover it. By looking at the parent function we learn it must be of length 64 and, as explained before, the inserted values must be valid hexadecimals (while letters after F are considered invalid, other symbols are not, but they don't seem to contribute to product key).

The legi ID is pushed on the stack early in `licence_check` function body, before calling 4 helper functions. These take as input parameter either the product key or both the legi number and the product key. The return values of the first two functions are involved in a bitwise `AND` (that is, they must both succeed). The result is involved in bitwise `AND` with return value of the third and so on. Put simply, every helper function performs some kind of validation, and they must all succeed.
In the end, if `licence_check` function return value is different from 0 then the product key is valid and activation is successful.

By debugging the program we may have clearer in mind the content of the stack, as the pseudocode produced by the decompiler can be a bit confusing at this point.

```
----legi-----------------------
 00007FFC98792980 0204000703030103
 ---------------------------------
 00007FFC98792988 6045BC9D2BD18100
 00007FFC98792990 0000000000000000
 00007FFC98792998 00005592FC6B3E4B
 00007FFC987929A0 0000000000000000
 ----product key------------------
 00007FFC987929A8 1267452371563402
 00007FFC987929B0 3412674523715634
 00007FFC987929B8 5634126745237156
 00007FFC987929C0 7056341267452371
 ---------------------------------
```

Every line is 8 bytes long. We can notice that *little endian* order is employed.

For this particular example, the product key was `02345671234567123456712345671234567123456712345671234567123456712345670`. Every digit is represented in hexadecimal, which requires 4 bits per value. Hence in first byte we should store 02, the first two digits of the product key. Since little endian order is used though, this is actually the last byte in first line.

The first line shows the legi number as loaded on the stack. Recall that the return value of `read_legi` was an integer, not a string.
Inside `licence_check` function the following operations are performed.

```
int i = 7
do {

legi_array_stack[i] = (char)(legi_tmp % 10);
 i--;
 legi_tmp = legi_tmp / 10;

 } while (i != -1);

 \*
 * legiID = 31337042
 * legi_array_stack[i--] = legi_tmp % 10 => 24073313
 *  + cast to char => 0204000703030103
 *\
```

Now we can dive deep in each of the helper functions.

1. The first helper function takes as input parameter the product key. Notice that the product key is stored in 4 vectors of 8 bytes (16 digits each). They are added together and the result is progressively accumulated in `RAX` register. The final result is compared with `FFD7C787879FC7FF`, hard coded value stored in `RDX` register. It is useful to also observe that only the least significant 8 bytes are retained, as expected given the size of the register (it is important to remember it when trying to generate a valid key). The return value of such equality is also the return value

of the function.

The code should look something like:

```
return productKey[0] + productKey[1] + productKey[3] +
    productKey[4] == 18435422984574322687;
```

2. The second helper function takes as input the legi number and the product key as loaded on the stack. This is what the code looks after some cleanup:

```
int i = 0;
do {

  int tmp = XOR_0xffffffbd(*(productKey_stack + i))
  int tmp2 = rotate_left(tmp);

  if (tmp2 != *(legi_stack + i))
    return FAILURE;

  i++;

} while (index != 8);

return SUCCESS;
```

`XOR_0xffffffbd` is taking 1 byte at a time (corresponding to 2 digits of the product key), to perform `XOR` with `0xffffffbd`. The lower byte of the returned value is rotated left (left circular shift) and the result should correspond to the current legi digit. By iterating 8 times, all the values in the legi are processed, whilst only the first 16 digits of the product key are under examination. If any of such checks fails, `FAILURE` (i.e. 0) is returned. Notice that we can draw 8 equations for 8 unknown bytes. That is, it is enough to solve the second helper function to get the first 8 bytes (16 digits) of the product key.

3. The third helper function takes as input only the product key. The cleaned up code of the function body is the following:

```
int i = 0;
do {

  if (*(productKey_stack + 15 - i) ^ *(productKey_stack + i)) !=
      i)
    return FAILURE;

  i++;

} while (i != 8);

return SUCCESS;
```

The first two vectors of 8 bytes of the product key are processed here. Take the first iteration: the `XOR` of the first value of the first vector with the last one of the second should not be equal to the offset (or position) of the first value. It works similarly for successive iterations, progressively increasing the offset on one side and decreasing the offset on the other. As we draw 8 more equations, given that by now we could already know the first 16 digits, we can solve for the 8 bytes of the second vector and get the next 16 digits of the product key.

4. The fourth helper function takes as input both the product key and the legi number. The check performed aims at finding a fibonacci sequence in the values of the third vector of the product key, where the first byte should carry a value equal to the sum of the figures in the legi. The code of such function can be simplified as follows:

```
long i;
 char sum;

 i = 0;
 sum = '\0';
 do {
   sum += *(legi_stack + i);
   i++;
 } while (i != 8);

 if (*(productKey_stack + 16) == sum) {

   i = 17;
   while (*(productKey_stack + i) ==
          *(productKey_stack + -2 + i) + *(productKey_stack + -1 +
              i))) {
     i++;
     if (i == 24) {
       return SUCCESS;
     }
   }
 }
 return FAILURE;
```

Finally, from the fourth helper we see that we can find the next 16 digits in the product key.

It is important to observe that only the first helper function imposes some constraints on the final 16 digits and thus its solutions constitutes the final step in our attempt to generate a valid product key.

- The logical conditions on the license check are too many and too convoluted to be solved by eye. We can use a *solver* like `z3`: we can program it in Python, and find a solution to the check. How does your model look like? Which features did you use from `z3`? What is the final product license you find for your executable?

Note: This last question is challenging.

Solution:

Thanks to the deep dive into the helper functions we know have a clear idea on how to model them easily enough. We can than use `z3` solver to get a valid product key. We don't need many special features (even though there probably exist some that would allow for a simpler solution than the one shown in the following). It is sufficient to declare a `z3` solver, add the suitable conditions and get the model found, provided the solver has managed to identify at least one possible solution. Having to deal with the notation of `z3` models has turned out to be quite cumbersome sometimes, which is why helper 1 and 2 are actually easier address without using the solver: for them it is sufficient to reverse two very simple equations.

Notice that helper1 can be solved separately to get the final 16 digits of the product key, since by solving the first three helpers we already have enough equations to get the first 48 digits. Indeed only the first helper impose some constraint on the last 16 values , as explained before.

In the following it is reported the code used to solve the task and get a valid licence.

```
import collections
import struct
import ast

import z3

legi = []
for i in [3, 1, 3, 3, 7, 0, 4, 2]:
    legi.append(i)

# each array represents 16 digits
prodKey1 = [z3.BitVec('prodKey_%s' % i, 8) for i in range(0, 8)]
prodKey2 = [z3.BitVec('prodKey_%s' % i, 8) for i in range(8, 16)]
prodKey3 = [z3.BitVec('prodKey_%s' % i, 8) for i in range(16, 24)]
prodKey4 = [z3.BitVec('prodKey_%s' % i, 8) for i in range(24, 32)]
# prodKey4 not used, since we need to deal with little endian, easier
    without solver (shown here only for clarity and completeness)

s = z3.Solver()


def ROR(n, bits):
    n = n & (2 ** bits - 1)
    b = n & 1
    n = n >> 1
    if b:
        n = n | (1 << (bits - 1))

    return n
```

```
# helper 2 (without z3 solver for convenience)
for i in range(0, 8):
    prodKey1[i] = ROR(legi[i], 8) ^ 0xbd

# helper 3
for i in range(0, 8):
    s.add(prodKey1[i] ^ prodKey2[7 - i] == i)

# helper 4
legi_sum = 0
for i in legi:
    legi_sum += i

s.add(prodKey3[0] == legi_sum)
s.add(prodKey3[1] == prodKey3[0] + prodKey2[7])

for i in range(2, 8):
    s.add(prodKey3[i] == prodKey3[i - 1] + prodKey3[i - 2])

# print model and store values appropriately
if s.check().r > 0:
    model = s.model()

    prodKey_map_tmp = ast.literal_eval(str(model).replace("=", ":").
                                    replace("[", "{\"").replace("]",
                                        "\"}").
                                    replace(" : ", "\" : \"").
                                    replace(",\n", "\",\""))

    prodKey_map_tmp2 = {}
    for key, value in prodKey_map_tmp.items():
        prodKey_map_tmp2[key.replace(' ', '').
        replace('_9', '_09').
        replace('_8', '_08')] = "0x{:02x}".format(int(value))

    prodKey_map_tmp3 =
        collections.OrderedDict(sorted(prodKey_map_tmp2.items()))

    d1 = dict(prodKey_map_tmp3.items()[:len(prodKey_map_tmp3) / 2])
    d2 = dict(prodKey_map_tmp3.items()[len(prodKey_map_tmp3) / 2:])

    prodKey2 = collections.OrderedDict(sorted(d1.items()))
    prodKey3 = collections.OrderedDict(sorted(d2.items()))

    # print results
    list_tmp = []
    for val in prodKey1:
        list_tmp.append(hex(val).replace('0x', ''))

    prodKey1 = hex(int("0x" + "".join(list_tmp), base=16)).rstrip("L")
```

```
print(prodKey1)
# => 0x3c3d3c3c3ebdbfbc

list_tmp = []
for val in prodKey2.values():
    list_tmp.append(val.replace('0x', ''))

prodKey2 = hex(int("0x" + "".join(list_tmp), base=16)).rstrip("L")
print(prodKey2)
# => 0xbbb9b83a3f3e3c3c

list_tmp = []
for val in prodKey3.values():
    list_tmp.append(val.replace('0x', ''))

prodKey3 = hex(int("0x" + "".join(list_tmp), base=16)).rstrip("L")
print(prodKey3)
# => 0x17536abd27e40bef

# helper 1 (without z3 solver for convenience)

# little endian
a1 = struct.pack('<Q', int(prodKey1, 16)).encode('hex')
a2 = struct.pack('<Q', int(prodKey2, 16)).encode('hex')
a3 = struct.pack('<Q', int(prodKey3, 16)).encode('hex')

CONST = 18435422984574322687 # from source code
a4 = hex(CONST - (int(a1, 16) + int(a2, 16) + int(a3, 16) &
    0x00ffffffffffffff))

# back to big endian (reverse again)
prodKey4 = struct.pack('<Q', int(a4.strip('L'), 16)).encode('hex')
print(prodKey4)
# => f17d4053e2e7cf17
```

The final product key which allows for successful activation is:
`3c3d3c3c3ebdbfbcbbb9b83a3f3e3c3c17536abd27e40beff17d4053e2e7cf17`

(both lower and uppercase work)
`3C3D3C3C3EBDBFBCBBB9B83A3F3E3C3C17536ABD27E40BEFF17D4053E2E7CF17`