Security Engineering

Balázs Hain

2022

1 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.

1.1 Solution

Since program was written in a way that other sequences from that point forward are in no connection with the function that does the licence check routine, meaning that the other sequences are not invoked by the check function or do not require any result that the check function could provide, it can be simply skipped by replacing the function call with a no-operation command. Other options would require finding that path in the control flow graph that lead to the authenticated state, and modifying the jump or compare actions in such way that it would lead us to that state. Given the number of jump actions that task would take more time than the other.

In the ARM64 instruction set NOP action corresponds to the `0xD503201F` opcode. First we should find the address of the `bl <keycheck>` call in the main function. From that address (`0x00100f94`), we should replace the next 4 bytes with the opcode of NOP operation in reverse order due to endianness. (`1F 20 03 D5`). Or we could have used the manual edit mode in Ghidra, which would have allowed us to edit the assembly (and write `nop` instead of the `bl <keycheck>` command), but that would have inherently lead to the same thing. Voilà, the program has been patched.

- 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?

1.2 Solution

The licence check consists of two steps, handled by 2 distinct functions: reading, validating and formatting the input, and checking the correctness of the licence. We are going the inspect the former function. It expects a licence number of 64 character length, so it reads the input into a 256 long character buffer, but only processes the the first 64 elements. Then this sequence is transformed by assigning a number n to each character c in the following way: $(n = t(c))$

- this process is not case-sensitive, thus $t(c) = t(uppercase(c)) = t(lowercase(c))$
- if c is a letter between a and f (or A and F), then $t(c) = ASCII(lowercase(c)) - ASCII(a) + 10$, which is the hexadecimal value between $0xA - 0xF$,
- if c is a letter between g and z (or G and Z), then c is an invalid character and it is omitted from the processing,
- if c is a number then $t(c) = ASCII(c) - ASCII(0)$ (it takes the value of the number),
- if c is any other character $t(c) = 255$

Therefore $t(c) \in [0, 15] \vee \{255\}$. The resulting sequence is stored in a 32 byte memory, so that 2 charters from the sequence, with values between 10-15, can fit 1 byte (since $16 = 2^4$ and 1 byte = 8 bit). This remark also suggests that special characters with value of 255 are thrown away.

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.

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

1.3 Solution

The operations in the licence check function can be divided into 4 "helper" functions, which are probably inlined functions due to their unique calling conventions. Some of these functions also have nested (inlined) functions in them. These licence check function (and their helper functions) mainly operate on two data structures: `char[8] legi_array` which contains the {'3', '1', '3', '3', '7', '0', '4', '2'} characters (the Legi that we have entered to access the the licence check), and `byte[32] checksum` the licence we entered, stored in a format as discussed above. Now, let's discuss the inner workings of these helper functions:

`bool fun1(int64[4] checksum)`: It takes the *checksum* variable as an argument, and divides 32 byte memory section into 8 byte ranges, as it were an int64[4] array. It returns true if the 4 consecutive 8 byte sections sum up to the value `0xFFD7C787879FC7FF`.

```
checksum[0] + checksum[1] + checksum[2] + checksum[3] == 0xFFD7C787879FC7FF
```

`bool fun2(byte[8] legi_array, byte[32] checksum)`: The following statement must hold for $\forall i : 0 \leq i < 8$ to the function to return true:

`legi_array[i] == ((checksum[i]` $\oplus$ `0xc9) > > 0x6` $\wedge$ `0x3)` $\vee$ `((checksum[i]` $\oplus$ `0xc9)` $\wedge$ `0x3f) < < 2`

, where `legi_array[i]` is a character of the entered Legi, and `checksum[i]` is the same as if we took the 2*i, and 2*i+1*th* licence value, transformed them according to the parser described above, stored each as 4 bit values and then concatenated them into a single byte. (1)

`bool fun3(byte[32] checksum):` If the following statement holds $\forall i$ $0 \leq i < 8$: `i == checksum[15 - i]` $\oplus$ `checksum[i]`, then the function returns with true else with false. Where `checksum[k]` is a byte data, realized in the same way as in (1).

`bool fun4(char[8] legi_array, byte[32] checksum)`: This function returns true if two statements hold:

1. the `checksum[16] ==` $\sum_i^8$ `legi_array[i]` (the 16^{th} byte of the converted licence value, which is the the 32nd and the 33rd character in the licence concatenated together to an 8bit vector, equals to the sum of the values of the characters in the legi_array), and
2. `checksum[i] == checksum[i - 1] + checksum[i- 2]` $\forall i : 17 \leq i < 24$, meaning from the 17th element to the 24th, elements in the checksum (interpreted as a char array) constitute a fibonacci sequence (based on the previous two elements).

`checksum[k]` is a byte long, and regarded as in (1).

If all 4 helper functions return true, the licence check succeeds, thus we can access the Lecturer Version.

- 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.

1.4 Solution

Since we operate on different representations of a byte sequence in the memory using bitwise operands, it is natural to represent our licence as bit vectors `BitVec`s. We should also take that into consideration, how the whole sequence is stored in the memory (endianness). Let $B_b = (l_{i1}, l_{li2})$ $(0 \leq i < 64, 0 \leq l_i < 16)$ be two consecutive parsed characters of the licence, which are stored in a byte (one on the upper and one on the lower 4 bits). According to the big-endian representation in ARM chipsets the licence is stored in the following way: the bytes are in reverse byte order, while $B_b = (l_{i1}, l_{li2})$ in each byte stored as $i_1 < i_2$. Eg.: If $l_{i1}, l_{li2}, l_{i3}, l_{li4}$, is a sequence in the licence in the same order as we typed the characters in, then they are stored in the memory as $B(l_{i3}, l_{li4}), B(l_{i1}, l_{li2})$.

Knowing that, we can use 4bit long bit vectors that represent each character in the licence, and construct the `data` function that yields `bits` amount of data from the `checksum + offset` address.

Then we can add all the constraints imposed by the helper functions to the Z3 solver, and eventually transform back each 4bit vectors to their original character representation. The solver yields the following licence: `0989090908c9c8494ececc0c0a0b8809172037578ee573589150921ae70d1354` which is indeed a correct licence and allows access to the Lecturer Edition.