Nahmii Token Audit

July 26th, 2019

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the Nahmii Token project. Firstly, we will provide a summary of our discoveries and secondly, we will show the details of our findings.

Summary

The contracts audited are from the Nahmii Token repository at https://github.com/hubiinetwork/nahmii-contracts/blob/v1.0.0/contracts/. The audit is based on the commit f1942e422411679f660b6d30fd1452451169bf92, and updated to reflect changes at f1942e422411679f660b6d30fd1452451169bf92.

The audited contracts are:

- NahmiiToken.sol: Defines ERC20 parameters of the token.
- RevenueToken.sol: The actual implementation of the token.

The following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.
- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.
- Wrong or missing error handling.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the function input requirements.

Detailed findings

Medium severity

Bad require in holdersByIndices creates integer overflow

The function holdersByIndices takes a range of indices in the form of two numbers, *up* and *low* and returns the holders inside that range. It also takes a boolean value indicating whether only the positive balance holders are required.
The function starts by requiring the *low* value being below the *up* value:

```
require(low <= up);
```

But then later caps the *up* value, which may invalidate that requirement:

```
up = up > holders.length - 1 ? holders.length - 1 : up;
```

If both low and up are greater than holders.length - 1, the lower value will become bigger than the upper value, bypassing the previous require statement.
Later if *posOnly* is false an integer overflow occurs, since *SafeMath* is not being used:

```
length = up - low + 1;
address[] memory _holders = new address[](length);
```

This will likely make length a huge value, and will consume all the gas of the transaction if called on chain. With the right parameters if the overflowed length ends up being small enough that it does go through, it will return an uninitialized array, since the for loop condition never holds.
In either case, while on itself it does not constitute an attack it is not a good behavior to have and should be fixed.
We recommend either capping the *up* value before the require takes place or capping the *low* value with the *up* value after capping the *up* value. The later does not need the require to be in place as it will be correct for any pair of numbers.

Minor severity

Holders array may grow too large

The array `holders` contains addresses that had any balance at any point in the token lifetime. Since this array never shrinks (0 balance addresses are not removed), it may grow too large. While accesses to this array are made in chunks which

eliminates the possibility of contract softlocking (DoS), the gas costs over time to parse the entire array may grow too large if done on chain.

Enhancements

Unnecessary ternary operator zero checks in *balanceBlocksIn*

The function checks for 0 using a ternary operator in this line:

```
r = (h == 0) ? 0 :
balanceBlocks[account][i].mul(h).div(balanceBlockNumbers[account][i].sub(1));
```

But it is unnecessary since the variable *h* being 0 will make the whole calculation 0. Even from a performance perspective the optimization is negligible since it's only calculated once. The function also checks for 0 here:

```
uint256 l = (i == 0) ? startBlock : balanceBlockNumbers[account][i - 1];
```

And while **it is necessary since otherwise `i - 1` will overflow**, it's not necessary for the calculation that comes after:

```
r = (h == 0) ? 0 :
balanceBlocks[account][i].mul(h).div(balanceBlockNumbers[account][i].sub(1));
```

If `i` is 0, balanceBlocks[account][i] should be equal to 0, according to the implementation of *addBalanceBlocks*:

```
uint256 length = balanceBlockNumbers[account].length;
balances[account].push(balanceOf(account));
if (0 < length)
   ...
else
    balanceBlocks[account].push(0);
```

And that will make the entire calculation 0, so running the calculation when `i` equals 0 wasn't needed in the first place.
Just to understand what we mean, here is an alternative implementation of that piece of code (Using the min function found in OpenZeppelin Math.sol):

At line 180, we need to initialize the return variable at 0 in case `i` is 0:

```
uint256 r = 0;
```

At lines 185 to 193 inclusive, we won't do the calculation if `i` is 0:

```
if (i > 0) {
    uint256 lowBlock = balanceBlockNumbers[account][i - 1];
    uint256 highBlock = balanceBlockNumbers[account][i];
    uint256 blockRange = highBlock.sub(lowBlock);
```

```
    uint256 partialRange = min(highBlock, endBlock).sub(startBlock);
    r = balanceBlocks[account][i].mul(partialRange).div(blockRange);
}
i++;
```

One letter variable names in function *balanceBlocksIn*

There are some variable names with only one letter that are not trivial indexes. This includes “l” (Which supposedly means lower block), “h” (Which supposedly means higher block) and “r” (Which supposedly means return value). It would be better to use the full names as it will make the code more clear at first glance.
Also “h” changes its meaning mid code here:

```
h = h.sub(startBlock);
```

It’s stops being a block number and becomes a block amount. In this case we recommend creating another variable for better clarity.

Unnecessary array length access in *balanceBlocksIn*

After the length is checked here:

```
if (balanceBlockNumbers[account].length == 0 || endBlock <
    balanceBlockNumbers[account][0])
    return 0;
```

It would be more clear to declare a lastIndex variable like this:

```
uint256 lastIndex = balanceBlockNumbers[account].length - 1;
```

And use that variable instead of length access through the code, swapping comparison operators for their strict version and vice versa when needed.

Conclusion

The concept of this token contract it’s pretty simple. We found the idea of calculating the area below the balance curve to be reasonable. Although it had some readability issues and a potential overflow, it is nothing critical that would put the contract in imminent danger.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the Nahmii project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.