ABDK CONSULTING

SMART CONTRACT AUDIT

Morphose

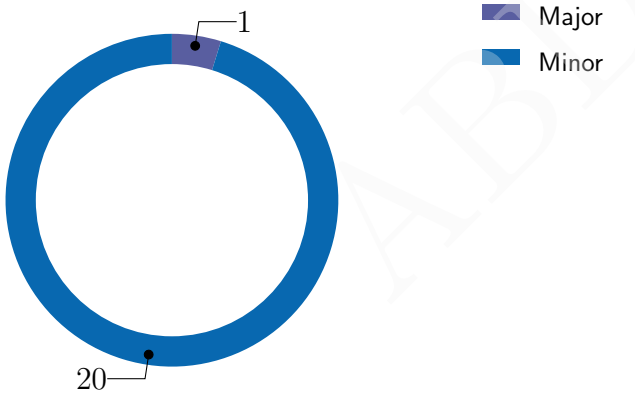
Solidity

abdk.consulting

SMART CONTRACT AUDIT CONCLUSION

by Mikhail Vladimirov and Dmitry Khovratovich
25th June 2021

We've been asked to review Morphose smart contracts given in separate files. We have found one major issue and a few moderate ones.



Findings

ID	Severity	Category	Status
CVF-1	Minor	Flaw	Opened
CVF-2	Minor	Readability	Opened
CVF-3	Minor	Procedural	Opened
CVF-4	Minor	Procedural	Opened
CVF-5	Minor	Readability	Opened
CVF-6	Minor	Suboptimal	Opened
CVF-7	Minor	Flaw	Opened
CVF-8	Minor	Unclear behavior	Opened
CVF-9	Minor	Bad datatype	Opened
CVF-10	Minor	Bad datatype	Opened
CVF-11	Minor	Flaw	Opened
CVF-12	Minor	Procedural	Opened
CVF-13	Minor	Flaw	Opened
CVF-14	Minor	Suboptimal	Opened
CVF-15	Minor	Readability	Opened
CVF-16	Minor	Suboptimal	Opened
CVF-17	Minor	Suboptimal	Opened
CVF-18	Minor	Suboptimal	Opened
CVF-19	Minor	Suboptimal	Opened
CVF-20	Major	Flaw	Opened
CVF-21	Minor	Suboptimal	Opened

Contents

1 Document properties

Version

Version	Date	Author	Description
0.1	June 24, 2021	D. Khovratovich	Initial Draft
0.2	June 24, 2021	D. Khovratovich	Minor revision
1.0	June 25, 2021	D. Khovratovich	Release

Contact

D. Khovratovich

khovratovich@gmail.com

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

We were given access to the Morphose repo. We audited only Solidity smart contracts:

- Morphose.sol;
- MembershipVerifier.sol.

We *did not* review the circuits for which proofs are verified.

2.1 About ABDK

ABDK Consulting, established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function. The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

2.2 Disclaimer

Note that the performed audit represents current best practices and smart contract standards which are relevant at the date of publication. After fixing the indicated issues the smart contracts should be re-audited.

2.3 Methodology

The methodology is not a strict formal procedure, but rather a collection of methods and tactics that combined differently and tuned for every particular project, depending on the project structure and and used technologies, as well as on what the client is expecting from the audit. In current audit we use:

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows code best practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places and that their visibility scopes and access levels are relevant. At this phase we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and is done properly. At this phase we understand user roles and permissions, as well as what assets the system ought to protect.

- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check that code actually does what it is supposed to do, that algorithms are optimal and correct, and that proper data types are used. We also check that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

3 Detailed Results

3.1 CVF-1

- **Severity** Minor
- **Status** Opened
- **Category** Flaw
- **Source** MembershipVerifier.sol

Recommendation A public function should require these parameters to be field elements. Otherwise, for example, value a = (0,FIELD_SIZE) would be accepted as valid as 'negate' converts the second coordinate to 0.

Listing 1:

```
211 uint[2] memory a,
    uint[2][2] memory b,
    uint[2] memory c,
    uint[3] memory input
```

3.2 CVF-2

- **Severity** Minor
- **Status** Opened
- **Category** Readability
- **Source** MembershipVerifier.sol

Recommendation This could be simplified to: 'return verify(inputValues, proof) == 0;'.

Listing 2:

```
224 if (verify(inputValues, proof) == 0) {
        return true;
    } else {
        return false;
    }
```

3.3 CVF-3

- **Severity** Minor
- **Status** Opened
- **Category** Procedural
- **Source** Morphose.sol

Description We didn't review this file.

Listing 3:

```
2 "./MerkleTree.sol";
```

3.4 CVF-4

- **Severity** Minor
- **Status** Opened
- **Category** Procedural
- **Source** Morphose.sol

Description This file wasn't provided.

Listing 4:

```
4 "./SafeMath.sol";
```

3.5 CVF-5

- **Severity** Minor
- **Status** Opened
- **Category** Readability
- **Source** Morphose.sol

Recommendation Consider using uint256 for field elements instead of bytes32.This would allow removing several type casts.

Listing 5:

```
13      bytes32 merkleRoot;
        bytes32 unitNullifier;
        bytes32[8] proof;

20  mapping(bytes32 => bool) public withdrawn;

25  event Deposit(bytes32 note, uint256 index, uint256 units);
    event Withdrawal(bytes32 unitNullifier);

38  function deposit(bytes32 note) public payable {

46      bytes32 leaf = merkleTree.hasher.poseidon([note, bytes32(
          ↪ units)]);

85      returns (bytes32[MERKLE_DEPTH] memory)

93  ) public pure returns (bytes32) {

112     bytes32[8] memory proof,
        bytes32 merkleRoot,
        bytes32 unitNullifier,
        bytes32 context
```

3.6 CVF-6

- **Severity** Minor
- **Status** Opened
- **Category** Suboptimal
- **Source** Morphose.sol

Recommendation Storing the whole tree on-chain is suboptimal. Common practice is to store only the root hash.

Listing 6:

```
19 MerkleTree.Data internal merkleTree;
```

3.7 CVF-7

- **Severity** Minor
- **Status** Opened
- **Category** Flaw
- **Source** Morphose.sol

Recommendation This storage variable should be declared as "immutable".

Listing 7:

```
21 uint256 public denomination;
```

3.8 CVF-8

- **Severity** Minor
- **Status** Opened
- **Category** Unclear behavior
- **Source** Morphose.sol

Description The meaning of this variable is unclear. It it really necessary?

Listing 8:

```
23 uint256 public anonymitySet;
```

3.9 CVF-9

- **Severity** Minor
- **Status** Opened
- **Category** Bad datatype
- **Source** Morphose.sol

Recommendation This argument should have type "Morph".

Listing 9:

```
29 address morphAddr,
```

3.10 CVF-10

- **Severity** Minor
- **Status** Opened
- **Category** Bad datatype
- **Source** Morphose.sol

Recommendation This argument should have type "MembershipVerifier".

Listing 10:

```
30 address verifierAddr,
```

3.11 CVF-11

- **Severity** Minor
- **Status** Opened
- **Category** Flaw
- **Source** Morphose.sol

Description There is no range check for this argument.
Recommendation Consider adding an explicit check that this argument is not zero.

Listing 11:

```
31 uint256 denomination_
```

3.12 CVF-12

- **Severity** Minor
- **Status** Opened
- **Category** Procedural
- **Source** Morphose.sol

Recommendation Consider adding require(units < BN128_SCALAR_FIELD) after this line for consistency.

Listing 12:

```
45 uint256 units = msg.value / denomination;
```

3.13 CVF-13

- **Severity** Minor
- **Status** Opened
- **Category** Flaw
- **Source** Morphose.sol

Description There is no way to reward the caller of the "withdraw" functions. This means that in the most cases the caller will be affiliated with the recipient. Such affiliation could be used to deanonymize the recipient, as in order to receive some ether from the protocol, the recipient would need to obtain some ether to pay gas for "withdraw" transaction.
Recommendation Consider adding an extra argument for the reward amount to be sent to 'msg.sedner'.

Listing 13:

```
51 }
```

3.14 CVF-14

- **Severity** Minor
- **Status** Opened
- **Category** Suboptimal
- **Source** Morphose.sol

Description The address of "withdrawn[args.unitNullifier]" is calculated twice.
Recommendation Consider calculating once and reusing.

Listing 14:

```
56      !withdrawn[args.unitNullifier],

70  withdrawn[args.unitNullifier] = true;
```

3.15 CVF-15

- **Severity** Minor
- **Status** Opened
- **Category** Readability
- **Source** Morphose.sol

Recommendation These two lines could be merged together: if (–currentUnits == 0) {

Listing 15:

```
71  currentUnits--;

73  if (currentUnits == 0) {
```

3.16 CVF-16

- **Severity** Minor
- **Status** Opened
- **Category** Suboptimal
- **Source** Morphose.sol

Recommendation The values "onePercent(denomination)" and "denomination - onePrecent(denomination)" should be calculated once in the constructor and stored in immutable variables.

Listing 16:

```
77 args.recipent.transfer(denomination - onePercent(denomination));
   payable(0x3e4Ff40a827d4Ede5336Fd49fBCbfe02c6530375).transfer(
      ↪ onePercent(denomination));
```

3.17 CVF-17

- **Severity** Minor
- **Status** Opened
- **Category** Suboptimal
- **Source** Morphose.sol

Description Hardcoding addresses is a bad practice.
Recommendation Consider passing the fee recipient address as a constructor argument and storing in an immutable variable.

Listing 17:

```
78 payable(0x3e4Ff40a827d4Ede5336Fd49fBCbfe02c6530375).transfer(
      ↪ onePercent(denomination));
```

3.18 CVF-18

- **Severity** Minor
- **Status** Opened
- **Category** Suboptimal
- **Source** Morphose.sol

Recommendation This function is overcomplicated. A 1% of a number rounded up could be calculated like this: (_value + 99) / 100;.

Listing 18:

```
97  function onePercent(uint256 _value) public view returns (uint256
       ↪ ) {
        uint256 roundValue = SafeMath.ceil(_value, 100);
        uint256 onePercent = SafeMath.div(SafeMath.mul(roundValue,
           ↪ 100), 10000);
100     return onePercent;
    }
```

3.19 CVF-19

- **Severity** Minor
- **Status** Opened
- **Category** Suboptimal
- **Source** Morphose.sol

Recommendation This is equivalent to SafeMath.div (roundValue, 100).

Listing 19:

```
99 uint256 onePercent = SafeMath.div(SafeMath.mul(roundValue, 100),
   ↪ 10000);
```

3.20 CVF-20

- **Severity** Major
- **Status** Opened
- **Category** Flaw
- **Source** Morphose.sol

Description There are no range checks for the parameters.
Recommendation Consider adding explicit checked that all the values are field elements.

Listing 20:

```
112 bytes32[8] memory proof,
    bytes32 merkleRoot,
    bytes32 unitNullifier,
    bytes32 context
```

3.21 CVF-21

- **Severity** Minor
- **Status** Opened
- **Category** Suboptimal
- **Source** Morphose.sol

Description Here arguments are repacked from a single array to several arrays, just to later be repacked into a Proof structure inside the 'verifyProof' function.
Recommendation Consider reducing the number of repacks.

Listing 21:

```
117 uint256[2] memory a = [uint256(proof[0]), uint256(proof[1])];
    uint256[2][2] memory b =
        [
120         [uint256(proof[2]), uint256(proof[3])],
            [uint256(proof[4]), uint256(proof[5])]
        ];
    uint256[2] memory c = [uint256(proof[6]), uint256(proof[7])];
    uint256[3] memory input =
        [uint256(merkleRoot), uint256(unitNullifier), uint256(
          ↪ context)];
```