ABDK CONSULTING

SMART CONTRACT
AUDIT

Weiroll

Weiroll

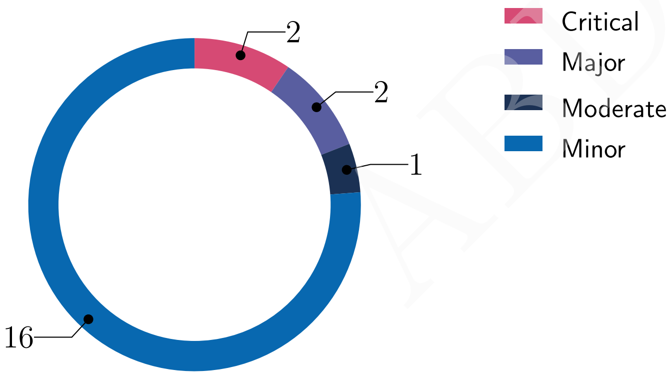
Solidity

abdk.consulting

SMART CONTRACT AUDIT CONCLUSION

by Mikhail Vladimirov and Dmitry Khovratovich
28th June 2022

We've been asked to review 6 files in a Github repository. We found 2 critical, 2 major, and a few less important issues. All critical and major issues have been fixed.



Findings

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

Contents

1 Document properties

Version

Version	Date	Author	Description
0.1	June 27, 2022	D. Khovratovich	Initial Draft
0.2	June 27, 2022	D. Khovratovich	Minor revision
1.0	June 28, 2022	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 have reviewed the contracts at repository:

- contracts/CommandBuilder.sol
- contracts/VM.sol

The fixes were provided in a new commit.

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
- **Category** Procedural
- **Status** Info
- **Source** CommandBuilder.sol

Recommendation Should be "^0.8.0" according to a common best practice, unless there is something special about this particular version. Also relevant for VM.sol.
Client Comment Disagree as there were patches to solc between 0.8.0 and 0.8.11 containing various bugfixes.

Listing 1:

```
pragma solidity ^0.8.11;
```

3.2 CVF-2

- **Severity** Minor
- **Category** Suboptimal
- **Status** Fixed
- **Source** CommandBuilder.sol

Description Defining top-level constants in a file named after a library makes it harder for a reader to find these constants.
Recommendation Consider either putting the constants inside the library, or moving them to a separate file named "constants.sol".

Listing 2:

```
uint256 constant IDX_VARIABLE_LENGTH = 0x80;
uint256 constant IDX_VALUE_MASK = 0x7f;
uint256 constant IDX_END_OF_ARGS = 0xff;
uint256 constant IDX_USE_STATE = 0xfe;
```

3.3 CVF-3

- **Severity** Minor
- **Category** Documentation
- **Status** Fixed
- **Source** CommandBuilder.sol

Description The meaning of these constants is unclear.
Recommendation Consider documenting.

Listing 3:

```
uint256 constant IDX_VARIABLE_LENGTH = 0x80;
uint256 constant IDX_VALUE_MASK = 0x7f;
uint256 constant IDX_END_OF_ARGS = 0xff;
uint256 constant IDX_USE_STATE = 0xfe;
```

3.4 CVF-4

- **Severity** Minor
- **Category** Readability
- **Status** Info
- **Source** CommandBuilder.sol

Description Uninitialized variables make code harder to read.
Recommendation Consider explicitly initializing with zero.
Client Comment Disagree, as solc 0-initializes variables, so not including the assignment "= 0" is less to read, thus more readable. It costs more in gas for this redundant zero-assignment.

Listing 4:

```
for (uint256 i; i < 32; i=_uncheckedIncrement(i)) {

for (uint256 i; i < 32; i=_uncheckedIncrement(i)) {
```

3.5 CVF-5

- **Severity** Minor
- **Category** Suboptimal
- **Status** Fixed
- **Source** CommandBuilder.sol

Description Replacing checked increment with a function call doesn't look like a good optimization.
Recommendation Consider using an unchecked block at the end of the loop body instead.

Listing 5:

```
for (uint256 i; i < 32; i=_uncheckedIncrement(i)) {

for (uint256 i; i < 32; i=_uncheckedIncrement(i)) {
```

3.6 CVF-6

- **Severity** Major
- **Category** Suboptimal
- **Status** Fixed
- **Source** CommandBuilder.sol

Description This loop basically calculates three things: 1. The "free" value 2. The "count" value 3. The "stateData" value if needed Items 2 and 3 are actually redundant. The "count" value is redundant, as Solidity memory model allows accumulating data in a newly allocated array without preliminary knowledge of the exact data amount. Just reserve a memory slot for the array length, write the data, then fill the length slot and advance the free memory pointer. The "stateData" value is redundant as the values could be copied from "state" to the result directly element by element, without allocating an intermediary flattened array.
Recommendation Consider refactoring the code to remove redundant logic.

Listing 6:

```
for (uint256 i; i < 32; i=_uncheckedIncrement(i)) {
```

3.7 CVF-7

- **Severity** Minor
- **Category** Suboptimal
- **Status** Info
- **Source** CommandBuilder.sol

Description This operation is performed in all branches.
Recommendation Consider doing at one place after the conditional operator.
Client Comment Disagree, as it costs more in gas to define and use intermediate variables in this case.

Listing 7:

```
    unchecked{free += 32;}

    unchecked{free += 32;}

unchecked{free += 32;}
```

3.8 CVF-8

- **Severity** Minor
- **Category** Suboptimal
- **Status** Fixed
- **Source** CommandBuilder.sol

Description The expression "add(add(ret, 36), count)" is calculated on every loop iteration.
Recommendation Consider just initializing "count" with the value "ret + 36".

Listing 8:

```
    mstore(add(add(ret, 36), count), free)

    mstore(add(add(ret, 36), count), free)

mstore(add(add(ret, 36), count), mload(add(statevar, 32)))
```

3.9 CVF-9

- **Severity** Minor
- **Category** Suboptimal
- **Status** Fixed
- **Source** CommandBuilder.sol

Description The expression "state[idx & IDX_VALUE_MASK]" is calculated several times.
Recommendation Consider calculating once and reusing.

Listing 9:

```
    uint256 arglen = state[idx & IDX_VALUE_MASK].length;

        state[idx & IDX_VALUE_MASK],

bytes memory statevar = state[idx & IDX_VALUE_MASK];
```

3.10 CVF-10

- **Severity** Minor
- **Category** Suboptimal
- **Status** Fixed
- **Source** CommandBuilder.sol

Description The expression "add(output, 32)" is calculated twice.
Recommendation Consider calculating once and reusing.

Listing 10:

```
argptr := mload(add(output, 32))

mstore(add(output, 32), sub(mload(output), 32))
```

3.11 CVF-11

- **Severity** Minor
- **Category** Unclear behavior
- **Status** Fixed
- **Source** CommandBuilder.sol

Description The returned value is ignored.
Recommendation Consider checking it for consistency.

Listing 11:

```
staticcall(
```

3.12 CVF-12

- **Severity** Minor
- **Category** Readability
- **Status** Fixed
- **Source** VM.sol

Description Defining top-level constants in a file named after a contract makes it harder for a reader to find these constants.
Recommendation Consider either putting the constants inside the contract, or moving them to a separate file named "constants.sol".

Listing 12:

```
uint256 constant FLAG_CT_DELEGATECALL = 0x00;
uint256 constant FLAG_CT_CALL = 0x01;
uint256 constant FLAG_CT_STATICCALL = 0x02;
uint256 constant FLAG_CT_VALUECALL = 0x03;
uint256 constant FLAG_CT_MASK = 0x03;
uint256 constant FLAG_EXTENDED_COMMAND = 0x80;
uint256 constant FLAG_TUPLE_RETURN = 0x40;

uint256 constant SHORT_COMMAND_FILL = 0x000000000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
```

3.13 CVF-13

- **Severity** Minor
- **Status** Info
- **Category** Readability
- **Source** VM.sol

Description Uninitialized variables make code harder to read.
Recommendation Consider explicitly initializing with zero.
Client Comment Disagree, as solc 0-initializes variables, so not including the assignment "= 0" is less to read, thus more readable. It costs more in gas for this redundant zero-assignment.

Listing 13:

```
for (uint256 i; i < commandsLength; i=_uncheckedIncrement(i)) {
```

3.14 CVF-14

- **Severity** Minor
- **Status** Fixed
- **Category** Suboptimal
- **Source** VM.sol

Description Replacing checked increment with a function call doesn't look like a good optimization.
Recommendation Consider using an unchecked block at the end of the loop body instead.

Listing 14:

```
for (uint256 i; i < commandsLength; i=_uncheckedIncrement(i)) {
```

3.15 CVF-15

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** VM.sol

Description The conversion from "bytes1" to "uint8" actually shifts the value, so there are two shifts here, while one shift would be enough.
Recommendation Consider calculating like this: flags = uint256(command » 216) & 0xFF;
Client Comment Disagree, in this case setting intermediate variable costs more in gas (have tried in this case many times)

Listing 15:

```
flags = uint256(uint8(bytes1(command << 32)));
```

3.16 CVF-16

- **Severity** Critical
- **Category** Flaw
- **Status** Fixed
- **Source** VM.sol

Description The expression "i++" returns the oritinal value of "i", not the increased value, so the same value will be used for "command" and "indices".
Recommendation Should probably be "++i" instead of "i++".

Listing 16:

```
command = commands[i];

      indices = commands[i++];
```

3.17 CVF-17

- **Severity** Minor
- **Category** Suboptimal
- **Status** Info
- **Source** VM.sol

Recommendation The conversion to "uint256" is redundant, as the bitwise "or" operator is able to work with the "bytes32" type.
Client Comment Disagree, as compiler disallows proposed change.

Listing 17:

```
indices = bytes32(uint256(command << 40) | SHORT_COMMAND_FILL);
```

3.18 CVF-18

- **Severity** Minor
- **Category** Suboptimal
- **Status** Fixed
- **Source** VM.sol

Description The expression "flags & FLAG_CT_MASK" is calculated several times.
Recommendation Consider calculating once and reusing.

Listing 18:

```
if (flags & FLAG_CT_MASK == FLAG_CT_DELEGATECALL) {
} else if (flags & FLAG_CT_MASK == FLAG_CT_CALL) {
} else if (flags & FLAG_CT_MASK == FLAG_CT_STATICCALL) {
} else if (flags & FLAG_CT_MASK == FLAG_CT_VALUECALL) {
```

3.19 CVF-19

- **Severity** Major
- **Category** Flaw
- **Status** Fixed
- **Source** VM.sol

Description There is no check that "v.length == 32".
Recommendation Consider adding such check.

Listing 19:

```
bytes memory v = state[uint8(bytes1(indices))];
```

3.20 CVF-20

- **Severity** Critical
- **Category** Flaw
- **Status** Fixed
- **Source** VM.sol

Recommendation Should be: calleth := mload (add(v, 0x20))

Listing 20:

```
mstore(calleth, add(v, 0x20))
```

3.21 CVF-21

- **Severity** Moderate
- **Category** Suboptimal
- **Status** Info
- **Source** VM.sol

Description This assumes that "outdata" is an encoded string error message, while it could also be an encoded named error or use some custom format.
Recommendation Consider packing "outdata" as is without trying to decode it.
Client Comment Outdata is bytes, so even if malformed, will not cause runtime error if cast to string

Listing 21:

```
outdata := add(outdata, 68)
```