ABDK CONSULTING

SMART CONTRACT
FINAL AUDIT

MakerDAO

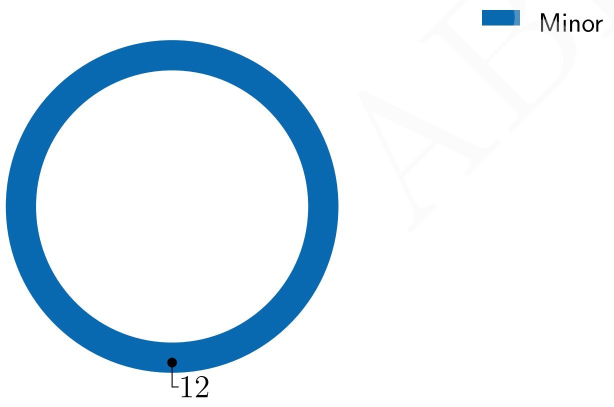
Crop. Phase 2

abdk.consulting

SMART CONTRACT AUDIT CONCLUSION

by Mikhail Vladimirov and Dmitry Khovratovich
6th May 2021

We've been asked to review a part of MakerDAO Crop smart contracts given in separate files. These contracts deal with dividend distribution from tokens deposited to Maker contracts. We have checked protocol correctness, optimality, scalability, and fairness. This is the second version of the code that we release. We have identified only minor issues in this version.



Findings

ID	Severity	Category	Status
CVF-1	Minor	Procedural	Opened
CVF-2	Minor	Suboptimal	Opened
CVF-3	Minor	Suboptimal	Opened
CVF-4	Minor	Unclear behavior	Opened
CVF-5	Minor	Procedural	Opened
CVF-6	Minor	Suboptimal	Opened
CVF-7	Minor	Suboptimal	Opened
CVF-8	Minor	Overflow/Underflow	Opened
CVF-9	Minor	Suboptimal	Opened
CVF-10	Minor	Suboptimal	Opened
CVF-11	Minor	Overflow/Underflow	Opened
CVF-12	Minor	Suboptimal	Opened

Contents

1 Document properties **5**

2 Introduction **6**

2.1 About ABDK 6

2.2 Disclaimer 6

2.3 Methodology 6

3 Detailed Results **8**

3.1 CVF-1 8

3.2 CVF-2 8

3.3 CVF-3 8

3.4 CVF-4 9

3.5 CVF-5 9

3.6 CVF-6 9

3.7 CVF-7 9

3.8 CVF-8 10

3.9 CVF-9 10

3.10 CVF-10 10

3.11 CVF-11 11

3.12 CVF-12 11

1 Document properties

Version

Version	Date	Author	Description
0.1	Apr. 10, 2021	D. Khovratovich	Initial Draft
0.2	Apr. 10, 2021	D. Khovratovich	Minor revision
1.0	Apr. 11, 2021	D. Khovratovich	Release
1.1	May. 5, 2021	D. Khovratovich	Issue fix
2.0	May. 6, 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 have reviewed the contracts in the crop repository, version 1.1.0, and have checked the difference to version 1.0.0:

- crop.sol;
- sushi.sol

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.
- **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** Opened
- **Source** sushi.sol

Recommendation This interface should be moved to a separate file.

Listing 1:

```
TimelockLike {
```

3.2 CVF-2

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

Description These parameters are probably redundant as their values could be derived from the "masterchef_" smart contract.

Listing 2:

```
address gem_,
address bonus_,
```

3.3 CVF-3

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

Description Is it really necessary to have two very similar functions?
Recommendation Consider merging them into one function accepting the "data" argument of type "bytes32" and converting it to proper type depending on the value of the "what" argument.

Listing 3:

```
function file(bytes32 what, uint256 data) external auth {

function file(bytes32 what, address data) external auth {
```

3.4 CVF-4

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

Description It looks weird that in some cases the full signature is hashed, while in the other cases only the selector is hashed.

Listing 4:

```
bytes32 txHash = keccak256(abi.encode(masterchef, value,
    ↪ signature, data, eta));
```

3.5 CVF-5

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

Recommendation This interface should be moved to a separate file to simplify code navigation.

Listing 5:

```
VatLike {
```

3.6 CVF-6

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

Description This type case would not be unnecessary if the "vat_" parameter would have type "VatLike".

Listing 6:

```
vat = VatLike(vat_);
```

3.7 CVF-7

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

Description "1e18 / toGemConversionFactor" would be cheaper.

Listing 7:

```
to18ConversionFactor = 10 ** (18 - dec_);
```

3.8 CVF-8

- **Severity** Minor
- **Category** Overflow/Underflow
- **Status** Opened
- **Source** crop.sol

Description Phantom overflow is possible here. The rounded up division itself never overflows, but this formula does overflow in some cases.
Recommendation Consider calculating like this: x == 0 ? 0 : (x - 1) / y + 1.

Listing 8:

```
z = add(x, sub(y, 1)) / y;
```

3.9 CVF-9

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

Recommendation Consider implementing a "muldivup(x, y, z)" function that calculated x * y / z rounded up.

Listing 9:

```
z = divup(mul(x, WAD), y);
```

```
z = divup(mul(x, y), RAY);
```

3.10 CVF-10

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

Recommendation require (-int256(wad)<0) would be better as it will allow wad=2̂255.

Listing 10:

```
require(int256(wad) > 0);
```

3.11 CVF-11

- **Severity** Minor
- **Category** Overflow/Underflow
- **Status** Opened
- **Source** crop.sol

Description Overflow is possible in the type cast.

Listing 11:

```
vat.slip(ilk, msg.sender, -int256(wad));
```

3.12 CVF-12

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

Description mul(cs, wad) / ss is equivalent to wdiv (cs, ss).

Listing 12:

```
uint256 dcrops = mul(cs, wad) / ss;
```