PUBLIC

Code Assessment

of the FlapperUniV2 Smart Contracts

June 06, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help MakerDAO with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of FlapperUniV2 according to Scope to support you in forming an opinion on their security risks.

Client implemented a new flapper contract. Rather than auctioning off the surplus DAI, it is now exchanged and added to an UniswapV2 pool.

The most critical subjects covered in our audit are functional correctness of the changed code and the impact of the change on the existing system.

It's worth noting that, by design, this new flapper spends up to x2.2 times the amount of DAI the Vow expects it to spend. For more details please refer to the informatinoal issue.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1
• Code Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the FlapperUniV2 repository based on the documentation files. The scope consists of the three solidity smart contracts:

1. ./src/FlapperMom.sol
2. ./src/FlapperUniV2.sol
3. ./src/OracleWrapper.sol

In Version 3 deployment scripts have been added:

1. ./deploy/FlapperDeploy.sol
2. ./deploy/FlapperInit.sol
3. ./deploy/FlapperInstance.sol

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	3 May 2023	a462de94dbc1e30186af9ee49813c813aad4a191	Initial Version
2	18 May 2023	b222ed94b447f14e4305ba34118f851db16daebe	After Intermediate Report
3	28 May 2023	f45c76691195ffb51c7d2bdab45db5a6316f459e	Deployment Scripts

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

UniswapV2 is not in scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

MakerDAO implemented a new flapper contract. Leveraging Uniswap v2, surplus DAI of the Vow is exchanged for Gem tokens. The acquired Gem tokens, along with a proportional amount of additional DAI drawn from the Vow are deposited back into the liquidity pool. The liquidity pool shares are minted to a predefined `receiver`.

FlapperUniV2

Upon deployment of the contract, immutables including the Gem token, pair (UniswapV2 pool) and the receiver are initialized.

The flapper contract features the following state variables that can be updated by priviledged roles:

- `pip` the reference price oracle used for the sanity check on the exchange rate of the swap.
- `hop` minimum amount of seconds between kicks.
- `want` relative multiplier (in unit of WAD) of the reference price to insist on during the price check.

Furthermore, there is a variable `live` which is set to `1` upon deployment and to `0` when the contract is caged during global shutdown.

Whenever the Vow of the DSS has sufficient surplus, anyone may execute `Flapper.kick()` through `Vow.flap()`. Before calling `Flapper.kick(bump, 0)`, the Vow ensures the debt is zero and that there is sufficient surplus. Note that this sufficient surplus check is circumvented by this new flapper as this flapper pulls more than the expected amount of DAI.

The contract's main function `kick()` works as follows:

- Pulls DAI from the `msg.sender` (Vow).
- Swaps this DAI to Gem tokens in the UniswapV2 pool. A sanity check on the exchange rate is done using an external oracle.
- Calculates and pulls an additional amount of DAI from the `msg.sender` (Vow) in order to add the received Gem tokens and DAI in a balanced manner to the pool.
- A sanity check on the DAI amount is done before the DAI and Gem tokens are transferred to the pool. The LP tokens are minted to the receiver.

The receiver will be a PauseProxy controlled by the Governance. LP tokens are internally called "Elixir".

Shutdown / Global settlement:

In step 1 of the shutdown procedure the End module will call `Vow.cage()` which in turn calls the flapper. Since this flapper holds no assets, no funds must be move, the contracts `live` status is simply disabled.

LP tokens at the `receiver` are considered lost for the system in case of Emergency Shutdown due to malicious governance.

Administrative functions to add/remove priviledged roles (`rely()` / `deny()`) and to update parameters `file()` exist.

FlapperMom

This contract is used to bypass the governance delay when disabling the flapper in an emergency.

Since MakerDAO contracts do not feature fine grained access control, any priviledged account (`ward`) may call any priviledged function on the contract. To expose certain priviledged functions only, Mom contracts are used: These smart contracts feature code which allows to call certain functions on the target contract only, hence restricting access.

FlapperMom must be given the ward role in FlapperUniV2. It exposes `stop()` which updates the `hop` parameter of the flapper to `type(uint256).max`. Consequently, executions of `kick()` will be paused until the value has been reset.

`stop()` is a priviledged function. Access control is determined through a DsAuth scheme: Either the contract itself or the owner can call the function. Furthermore, if an authority contract is set, the authority contract is queried whether to grant access for this `msg.sender()` on this function.

Finally the contract offers functionality for the owner to update the owner (`setOwner()`) and the authority contract (`setAuthority()`).

OracleWrapper

This contract facilitates the conversion between an MKR/USD oracle and the USD value of the new governance token. The denomination of the new token is that 1 MKR token is equivalent to 1200 of the new governance tokens.

2.2.1 Trust Model and Roles

The Gem token used will be a rebranded version of the MKR token, the denomination is intended to be 1 MKR will be worth 1200 of this token.

With this flapper, surplus DAI is no longer auctioned off for MKR tokens. Instead, DAI is exchanged for Gem tokens and deposited into the pool. All lp pool shares minted are transferred to an external recipient set upon deployment. This recipient is intended to be a pause proxy controlled by the Governance.

The Governance is fully trusted to set all parameters honestly and correctly. The file functions feature no sanity checks, the assumption is that the governance thoroughly checks the parameters before the transaction is executed.

Callers of `Vow.flap()` which invokes `Flapper.kick()` are untrusted.

Uniswap V2 is expected to work correctly as documented. The pool is expected to have sufficient liquidity, an illiquid pool may have negative consequences for this flapper. And it is assumed the exchange rate on uniswap v2 is synced with other markets.

Wards of FlapperMom and the price feeds are fully trusted.

2.2.2 Changes in Version 2

- Flapper will first call `sync()` in case there is a donation to the pair and use the updated balance for swap and deposit later.
- Now the total amount of surplus for swap and donation is computed and pulled only once from Vow which saves one external call to vat and daiJoin.
- the `Kick` event is modified to show the swapped surplus and Gem token amount, the total amount of surplus used, and the liquidity minted.

2.3 Deployment Scripts

After the review of the main contracts deployment scripts have been added in Version 3. These are libraries which contain functions facilitating the deployment and initialization of the contracts.

FlapperDeploy: Library implementing functions to deploy the contracts. These are intended to be used locally with foundry in order to deploy the contracts.

- `deployFlapperUniV2()`: Deploys a new instance of FlapperUniV2 and FlapperMom with the given parameters. Ownership of both contracts is transferred to the given owner. The deployer retains no priviledged role afterwards.
- `deployOracleWrapper()`: Deploys a new instance of OracleWrapper with the given parameters. This contract features no priviledged roles.

FlapperInit: Library implementing functions to initialize the contracts. This library is intended to be used in a contract which is later executed as delegatecall in the priviledged pause proxy of the Governance.

- `initFlapperUniV2()`: After performing sanity checks, the initial values are set in the flapper and the vow. Adds `vow` and `mom` as wards. MCD_ADM fetched from the chainlog is set as the mom's authority and the chainlog is updated for MCD_FLAP and FLAPPER_MOM.
- `initOracleWrapper()`: Calls `kiss` on the `pip`, this adds the wrapper as a bud ensuring it can read from the pricefeed. Finally the address is set in the chainlog.

Despite the sanity checks, the functions rely on the parameters passed by the Governance to be correct.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Pull DAI From Vow All at Once Code Corrected

6.1 Pull DAI From Vow All at Once

Design Low Version 1 Code Corrected

CS-MUF-003

During a `kick()` call, two operations (a swap and a mint) on the UniswapV2 pair are executed consecutively. For each operation, an external call to the vat and daiJoin are invoked beforehand to pull the required DAI. However, the pool state after the swap can be precomputed, which means the total amount of DAI needed can be precomputed as well. It might be worth to do this to reduce the gas used and hence make the transactions slightly cheaper.

Code corrected:

The amount of DAI is now precomputed and pulled once.

6.2 Incorrect Comment

Informational Version 1 Specification Changed

CS-MUF-002

The comment `997 is the Uniswap LP fee` in `_getAmountOut()` is incorrect. `99.7%` represents the amount after deducting the fee, and the fee is `0.3%`.

```
function _getAmountOut(uint256 amtIn, uint256 reserveIn, uint256 reserveOut)
internal pure returns (uint256 amtOut) {
    uint256 _amtInFee = amtIn * 997; // 997 is the Uniswap LP fee
    amtOut = _amtInFee * reserveOut / (reserveIn * 1000 + _amtInFee);
}
```

Specification changed:

The incorrect comment has been removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Revert Reason When FlapperMom Stops Flapper

Informational **Version 1**

CS-MUF-001

FlapperMom can inhibit FlapperUniV2 in an emergency. It does so by setting the minimum time between two executions of `kick()` to `type.max(uint256)`. `kick()` will then revert due to the addition overflow:

```
require(block.timestamp >= zzz + hop, "FlapperUniV2/kicked-too-soon");
```

Except when `kick()` has never been executed before and `zzz` is still equal to 0, the require statement will cause the revert and emit the error MessageChannel.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 More Than `bump` Amount of DAI Used

Note **Version 1**

The Vow contract has been designed and documented with the original Flapper auctioning surplus DAI for MKR tokens in mind.

```
// Surplus auction
function flap() external returns (uint id) {
    require(vat.dai(address(this)) >= add(add(vat.sin(address(this)), bump), hump), "Vow/insufficient-surplus");
    require(sub(sub(vat.sin(address(this)), Sin), Ash) == 0, "Vow/debt-not-zero");
    id = flapper.kick(bump, 0);
}
```

By design, the new FlapperUniV2 may utilize up to 2.2 times the bump amount. The Vow contract may not anticipate the Flapper using more than the bump amount of DAI.

Depending on the values set for `bump` and `hump`, this could result in the Vow contract unexpectedly holding less than `hump` (surplus buffer) amount of DAI after a call to `kick()`, or a call to `kick()` unexpectedly reverting if the required amount of DAI is not available.

This behavior is now described in the README.

8.2 Unexpected Pair State

Note **Version 1**

Generally it is assumed that the free market ensures the pair represents the current market rate. However this can not be relied on as the state of the pair might be changed just before calling `kick()`. There are various possibilities why the pair could be in a state not matching the current market rate. Notably e.g. in case there is an unaccounted donation of tokens in the Uniswap pool (balance > reserve), the flapper will first call `sync()` on the pair and swap on the updated balances afterwards. This state can also be reached by an attacker donating and calling sync directly. Furthermore the state may be changed by trading.

Generally the possible manipulation is bounded by the following checks:

- In case the swapping ratio deviates too much from the reference price feed, `kick()` will revert.
- In case the liquidity of the pool is too shallow and the amount of surplus deposited back goes over 120% of swapped, `kick()` will also revert.

In theory, the following manipulations by donations are possible:

- One can donate within the price tolerance `want` to make the flapper trade at a bad price.
- One can intentionally donate to revert a `kick()` by pushing the price out of the price tolerance `want`.
- One can also donate to increase the liquidity and make a `kick()` which was going to revert (deposited larger than 120% of swapped) succeed.

MakerDAO is aware and adds the following considerations:

```
* One can donate within the price tolerance want to make the flapper trade at a bad price - the assumption
is that any trade above `want` is viable. It is of course possible for anyone to move the price with a swap,
which is probably even more economical than a donation. As long as `want` and `lot` are set correctly both
type of attempts should not be economical and are of course known limitations of a permissionless system.

* One can intentionally donate to revert a kick() by pushing the price out of the price tolerance want - same
as above, this can happen with a swap and is a known given. Keepers can use flashbots to avoid it.

* One can also donate to increase the liquidity and make a kick() which was going to revert (deposited larger
than 120% of swapped) succeed - if the kick succeeds it is intended behavior.
```