PUBLIC

# Code Assessment

of the Multiply

Smart Contracts

September 7, 2021

Produced for

by

# Contents

# 1 Executive Summary

Dear Oasis Team,

First and foremost we would like to thank you for giving us the opportunity to assess the current state of your Multiply system. This document outlines the findings, limitations, and methodology of our assessment.

The introduction of new functionality to skip the flashloan was inspected alongside the responses to the intermediate report. The new functionality significantly increased the code complexity. The parameter deciding whether a flashloan has to be taken allows packing two different functionalities into one method. Since the non-flashloan and flashloan cases differ enough, an option with two separate smart contracts could be choosen, allowing not only modularity of smart contract updates but also increasing the efficiency.

No security issues were uncovered while reviewing the stateless contracts of the MultiplyProxyActions system. However, some correctness issues were discovered:

- withdrawCollateral is added to borrowCollateral
- Incorrect decimals when handling collateral

Many issues were introduced with code added to support the new functionality. That highlights the importance of exhaustive testing especially as some of the new issues could have been detected by appropriate tests for the new functionality. The issue No check on amount received from flash loan illustrates how a fix may break a main functionality and can still remain undetected.

For a complete list of issues please refer to the overview of the findings.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

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 | 8 |
| • Code Corrected | 5 |
| • Specification Changed | 2 |
| • Risk Accepted | 1 |

| Low-Severity Findings | 22 |
|---|---|
| • Code Corrected | 9 |
| • Specification Changed | 1 |
| • Code Partially Corrected | 3 |
| • Risk Accepted | 3 |
| • Acknowledged | 6 |

# 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 `MultiplyProxyActions.sol`, `Exchange.sol` and `ExchangeData.sol` inside the Multiply repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 12 July 2021 | 026d2a61a43c7e660033bb3676f51b35f0825905 | Initial Version |
| 2 | 8 August 2021 | 57ff24efe3376fd3a77f37e49aa7eeb8b68bf2d4 | After Intermediate Report |
| 3 | 7 September 2021 | 10cfe4f37e9c5b9c5456e967b967fa623e03f632 | Final Commit |

For the solidity smart contracts, the compiler version `0.7.6` was chosen.

# 3 System Overview

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

Oazo provides a frontend for interacting with the Maker system which allows users to easily open a vault, deposit collateral and generate `DAI` backed by the locked collateral. Each user first deploys a `DSProxy` contract which is used to interact with the functionality provided. The proxy allows the user to execute code of the Oazo smart contracts aggregating functionality to perform certain actions wrapped in one transaction.

Like leverage trading that creates a larger position from a smaller investment amount, it is possible to use borrowed `DAI` from locked collateral to buy more collateral and use this collateral to borrow more `DAI`. By doing this repeatedly, long chains of exposure to the collateral can be generated. The new `MultiplyProxyActions` introduces the support for leverage actions while reducing the number of total transactions to one and the total number of transfers to the vault to one deposit by leveraging flash loans. Oazo's new `MultiplyProxyActions` contract contains functionality allowing users to easily increase and decrease the multiply factor and, thus, simplifies actions for creating, withdrawing and modifying leveraged positions. The core process can be described in three steps:

1. All values for the desired exposure / multiply factor are precomputed by the front end.
2. The respective function on the stateless smart contract is called and the parameters passed contain all necessary information.
3. The required large amount of collateral can be deposited all at once by leveraging flash loans. Therefore, `DAI` is generated only once. However, much more `DAI` will be available now but will be also needed to pay back the flash loan.

Effectively, gas costs are significantly reduced since there is no repetition of the traditional leverage actions.

The flash loan is always taken in `DAI` and swapped into collateral if needed. When reducing the multiply factor some collateral can be withdrawn from the vault. To repay the debt incurred by the flash loan, collateral is swapped into `DAI`. An exchange connector is deployed to support swapping `DAI` to collateral, and vice versa.

Following functions are provided by the smart contract:

**Increasing Exposure**

`openMultiplyVault` Opens a new vault for the user before calling `increaseMultipleDepositCollateral`.

`increaseMultipleDepositCollateral` Allows a user to increase the multiply factor on his vault while depositing collateral.

`increaseMultipleDepositDai` Allows a user to increase the multiply factor on his vault while depositing `DAI`.

`increaseMultiple` Allows a user to increase the multiply factor on his vault without depositing additional funds.

**Decreasing Exposure**

`decreaseMultiple`

Allows a user to decrease the multiply factor on his vault without withdrawing funds.

`decreaseMultipleWithdrawCollateral`

Allows a user to decrease the multiply factor on his vault while withdrawing collateral.

`decreaseMultipleWithdrawDai`

Allows a user to decrease the multiply factor on his vault while receiving the withdrawn collateral swapped into `DAI`.

`closeVaultExitCollateral`

Allows a user to close his vault (return all his debt) and withdraw the collateral.

`closeVaultExitDai`

Allows a user to close his vault (return all his debt) and receive the withdrawn collateral swapped into `DAI`.

**Execution:**

Changing the multiply factor on a user's vault of a certain `ilk` (collateral type) is done as follows:

Through his `DSProxy`, the user executes the appropriate function of the `MultiplyProxyAction` contract. Note that this executes the code of the `MultiplyProxyAction` in the context of the user's `DSProxy` contract since it is being executed as `DELEGATECALL`. All data required for the actions on the vault and the exchange are precomputed off chain by the front-end and passed as function parameters.

For the Vault action, all information related to the collateralized debt position is passed:

- `address gemJoin`: The Token Adapter for this collateral in the Maker system
- `address payable fundsReceiver`: The address withdrawn funds are to be transferred to
- `uint256 cdpId`: The unique id of the collateralized debt position in the cdp manager. Note that vaults not opened through the CdpManager are not supported. If the vault opening action is called, then the ID is zero.
- `bytes32 ilk`: The identifier of the vault type for the collateral. When the execution enters the code of MultiplyProxyActions, this is set to zero. It is set in the smart contract.
- `uint256 requiredDebt`: Amount of `DAI` required / to be flashloaned

- `uint256 borrowCollateral`: Amount of collateral needed to perform the action on the vault (will be swapped from the flashloaned `DAI`)
- `withdrawCollateral`: Amount of collateral to be withdrawn from the vault
- `withdrawDai`: Amount of `DAI` to be withdrawn from the vault
- `depositDai`: Amount of `DAI` to be deposited to the vault
- `depositCollateral`: Amount of collateral to be deposited into the vault.

These values must be initialized as required, not all values are required for all actions.

For the token swaps, more parameters are required:

- `address fromTokenAddress`: Address of the source token, either `DAI` or the collateral
- `address toTokenAddress`: Address of the tokens the source token is to be exchanged into, either `DAI` or the collateral
- `uint256 fromTokenAmount`: Amount of the source token, in respective tokenwei
- `uint256 toTokenAmount`: Amount of the target token, in respective tokenwei
- `uint256 minToTokenAmount`: Minimum amount of the target token to be received after the exchange
- `address exchangeAddress`: Address of the exchange to be used
- `bytes _exchangeCalldata`: Calldata for the call to the exchange

Addresses to be interacted with are passed as an address registry. This struct contains following data:

- `address jug`: Contract of the Maker system for the stability fee
- `address manager`: `CDPManager` contract address
- `address multiplyProxyActions`: The `MultiplyProxyActions` contract address
- `address aaveLendingPoolProvider`: The AaveLendingPoolProvider is used to query the address of Aave's flash loan contract
- `address feeRecepient`: (unused) the fee recepient
- `address exchange`: The address of the exchange contract

If necessary, setup actions such as opening a new vault or calculating the debt to pay when closing a value is done. If required by the chosen action, funds are transferred from the `DSProxy` to the `MultiProxyAction` account. Next, allowance to operate on the vault is given to the `CDPManager` and the call for the flash loan is crafted and being called. The funds are made available and the callback to function `executeOperation()` in the `MultiplyProxyActions` contract is made. Note that at this stage, the execution is no longer in the context of the `DSProxy` which executed a `DELEGATECALL`, but is in the context of `MultiplyProxyActions` since the callback from the flash loan provider executes the code of `MultiplyProxyActions` as a normal call.

Depending on the chosen action, the multiply factor will be either increased or decreased. In case of an increase, more debt in the vault is created and `DAI` is generated and is used to pay back the flash loan debt while the remaining generated `DAI` is either swapped to collateral and transferred back or directly transferred back to the `fundsReceiver`. In case of a decrease action, depending on the action, the collateral received is either partially swapped so that the flash loan can be paid back and the remaining collateral can be returned to the `fundsReceiver`, or completely swapped to pay off the flash loan debt and to return `DAI` to the `fundsReceiver`. Note that during token swaps, a fee is paid to a fee recipient.

Once the callback has finished and the flash loan is paid back, the context switches back to the `DSProxy`. If the flash loan has not been returned, the transaction will revert. Finally, the allowance given to the `CDPManager` is removed.

## 3.1 Roles & Trust Model

**User:** Untrusted, any user may interact with the `MultiplyProxyActions` contract to perform action on his own vault. Each user is responsible for the correctness of the input parameters passed to the function. The user may use the official frontend and trust it to aggregate all values correctly. That also includes trusting the third-party APIs used by the Oazo front-end (e.g., 1inch API generating swap request data).

**CdpManager:** Trusted, the `CDPManager` contract of Maker. During the action allowance is given to the `CDPManager` to manipulate the vault of the user.

**Oazo:** Owner of the smart contracts. Operates the Frontend used by most users to interact with the smart contract. The Frontend aggregates all values for the operation of the smart contract.

**Exchange (system contract):** Fully trusted. Called by the `MultiplyProxyActions` contract to facilitate the token swap via the external third-party exchange. Collects the fee for the system. Ensures that the minimum expected token amount has been returned by the external exchange.

**Exchange (third party):** `1inch` swap aggregator is to be used. Untrusted, with the exception that we assume it does not reenter into `MultiplyProxyAction`. The system exchange contract provides allowance for the source token and ensures that after the call a minimum amount of the tokens has been exchanged.

**FlashLoan Provider:** Fully trusted. Called from the proxy context after preparing the action, calls back into `MultiplyProxyActions.executeOperation()` passing the arguments to execute the operation. Fully trusted to do so honestly and correctly. Notably this includes that it does not reenter any other public function of the `MultiplyProxyActions` contract. In this version of the smart contract Aave v2 is used.

## 3.2 Changes in Version 2

- It's now possible to skip the flashloan in case no extra funds are required to execute the multiply action
- The top level function called when entering MultiplyProxyActions is now tracked and emitted in an Event

## 3.3 Changes in Version 3

- In the Exchange contract access control has been removed from the swap functions. This is necessary for the skipFL functionality to work.

# 4 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 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 associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. 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 the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

# 5 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 severities. 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.

# 6 Findings

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

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

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 | 1 |

- IERC20 Incompatible With Tether Risk Accepted

| | |
|---|---|
| Low-Severity Findings | 12 |

- Flash Loan Preparation When Skipping Flashloan Acknowledged
- Unnecessary Execution of Code Acknowledged
- Approval for Full Balance Instead of Amount Transferred Acknowledged
- Avoid Repeated Calls Code Partially Corrected Acknowledged
- Breaks for Collaterals With More Than 18 Decimals Risk Accepted
- Event Issues Code Partially Corrected Acknowledged
- Repeated Identical Multiplications Acknowledged
- Unchecked Return Values and Non-Compliant IGem Interface Risk Accepted
- Unnecessary Call of getAaveLendingPool Acknowledged
- Variables Could Be Immutable and Constant Code Partially Corrected
- _getWipeDart() Returns art When Closing Acknowledged
- transfer Is Used for ETH Transfers Risk Accepted

## 6.1 `IERC20` Incompatible With Tether

Design Medium Version 1 Risk Accepted

USDT, one of the collaterals, is not fully ERC20 compliant, notably some features lack the mandatory return value to comply with the standard.

In `MultiplyProxyActions` the interface `IERC20` is used to interact with the token contracts. In the interface definition a return value is expected for e.g., the `approve` and the `transfer` functions. Hence the Solidity compiler generates bytecode that expects a return value and reverts if there is none.

---

**Risk accepted:**

Oazo Apps Limited replied:

```
Currently, Tether is not used as collateral in Maker Protocol.
In case Tether is onboarded to Maker Protocol,
the multiply feature will be disabled for it.
```

## 6.2 Flash Loan Preparation When Skipping Flashloan

Design Low Version 2 Acknowledged

In version two, flashloans can be skipped. However, some parameters for taking the flashloan are prepared even though they will remain unused. More specifically, the `assets`, `amounts` and `modes` arrays are set which increases gas cost of the actions skipping flashloans. That part of flashloan preparation could be moved to the new `takeAFlashloan` function which would also further reduce the size of the code.

---

**Acknowledged:**

Oazo Apps Limited replied:

```
This is a minor gas inefficiency, will be fixed after MVP.
```

## 6.3 Unnecessary Execution of Code

Design Low Version 2 Acknowledged

`_getDrawDart()` returns the change in debt needed in order to have access to the DAI amount specified in paramter `wad`.

This function executes two external calls which could be omitted under certain conditions:

- In case the DAI amount needed is 0 the function the function could return 0 immediately
- In case enough DAI is available already, the call to `Jug.drip()` could be omitted.

Note that with the new skip flashloan functionality the case that enough DAI is already available will happen frequently and hence the code should be optimized for it.

---

**Acknowledged:**

Oazo Apps Limited replied:

```
This is a minor gas inefficiency, will be fixed after MVP.
```

## 6.4 Approval for Full Balance Instead of Amount Transferred

Design Low Version 1 Acknowledged

In `MultiplyProxyActions._closeWithdrawDai()` the exchange contract is approved to transfer the full token balance of the `MultiplyProxyActions` contract. The exchange contract however uses this approval only to transfer `ink` amount of collateral.

---

**Acknowledged:**

Oazo Apps Limited replied:

```
Exchange contract is fully trusted by the multiply proxy actions contract.
```

To the question why not a "infinite" (`uint256(-1)`) approval is given once for all executions to the exchange contract, Oazo Apps Limited responded:

```
Infinite approval could cause unexpected implications, therefore it is skipped in the scope of the MVP;
the only negative impact is gas inefficiency.
```

*Note: This issue was raised and discussed in Version 1 of the code reviewed where the skipFL functionality did not yet exist. After the introduction of the skipFL functionality access control to the swap functions of the Exchange contract has been removed, hence the trust model has changed.*

## 6.5 Avoid Repeated Calls

Design Low Version 1 Code Partially Corrected Acknowledged

In `MultiplyProxyActions.joinDrawDebt()`, `IManager(manager).urns(cdpData.cdpId)` is called twice in succession. Caching the address of the urn would be more efficient.

Additionally, `IManager(manager).urns(cdpData.cdpId)` is called three times for `closeVaultExitCollateral()` and `closeVaultExitDai()`. For both, the first call is in `closeVaultExitGeneric()` and the third call is in `wipeAndFreeGem()`. The second call is in `_closeWithdrawCollateral()` and `_closeWithdrawDai()` respectively. Similarly, the same holds for `IManager(manager).vat()`.

Moreover, `IVat(vat).hope(DAIJOIN)` is called on every increase action in the context of MultiplyProxyActions in the `joinDrawDebt` function. However, the opposite function `nope` is never called, meaning that `DAIJOIN` will always be able to modify the gem or DAI balance of MultiplyProxyActions. As this behaviour is required and the permission is never revoked, it could be called only once in the constructor to reduce the amount of needed gas.

---

**Code partially corrected:**

The occurence in `joinDrawDebt()` was optimized by caching the value. The calls to `IManager(manager).urns(cdpData.cdpId)` during the flow of closing and exiting have been reduced from three to two. For the last reported repeated call, `IVat(vat).hope(DAIJOIN)` the code remained unchanged.

**Acknowledged:**

Oazo Apps Limited replied for the unchanged sub-issues:

```
This is a minor gas inefficiency, will be fixed after MVP.
```

## 6.6 Breaks for Collaterals With More Than 18 Decimals

Design Low Version 1 Risk Accepted

Function `convertTo18` converts the amount into wad as follows:

```
wad = amt.mul(10 ** (18 - IJoin(gemJoin).dec()));
```

For tokens with more than 18 decimals, the subtraction results in an underflow. The subsequent exponentiation and multiplication likely overflow resulting in a random value returned.

---

**Risk accepted:**

Oazo Apps Limited replied:

```
Such approach is consistent with the current Multi-Collateral DAI
implementation and does not affect any existing collateral.
```

## 6.7 Event Issues

Design Low Version 1 Code Partially Corrected Acknowledged

There are several issues regarding events:

1. The callback function `executeOperation` in MultiplyProxyActions.sol contains the logic for using the flash loan. It emits the event `FLData`, logging how much was borrowed and how much is to be returned to Aave. It is emitted as follows:

   ```
   emit FLData(IERC20(DAI).balanceOf(address(this)),borrowedDaiAmount);
   ```

   While `borrowedDaiAmount` actually specifies the value that must be returned to Aave, the balance of the MultiplyProxyAction contract may not be the amount that was borrowed since the method `increaseMultipleDepositDai` may also increase the balance.
2. Event `FLData` in MultiplyProxyActions.sol indexes the amount borrowed and the amount to be returned. While in general indexing events is useful, there is no direct use of indexing this event and, thus, the cost of emitting this event could be reduced.
3. Event `AssetSwap` in Exchange.sol is unindexed. It could be helpful to index the assets associated with this event.
4. Event `FeePaid` in Exchange.sol does not log the beneficiary. This issue is related to issue feeRecepient of AddressRegistry and how it is resolved. If the receiver of the fee is not fixed for the contract, then logging the beneficiary could be useful since the beneficiary of the fee may change.

---

**Code partially corrected:**

1. Corrected: The event does not emit the balance anymore. Now, the difference between the balance and `depositDai` is emitted.
2. Corrected: `FLData` is now unindexed.
3. Corrected: `assetIn` and `assetOut` are indexed.
4. Not corrected: `FeePaid` now logs the beneficiary. Since the `feeBeneficiary` is public and cannot be changed, it is not necessary to log the beneficary. Gas costs could be reduced.

However, Oazo Apps Limited responded to that last point as follows:

```
Such approach allows us to immediately find fee beneficiary for every future instance of exchange;
it results in a minor gas inefficiency.
```

## 6.8 Repeated Identical Multiplications

Design | Low | Version 1 | Acknowledged

Some multiplications are done repeatedly within the same functions. E.g., in `_getDrawnDart()`, `wad.mul(RAY)` is calculated three times. Avoiding repeated multiplications and calculating it only once and storing the result may be favorable.

---

**Acknowledged:**

Oazo Apps Limited replied

```
The outcome of the proposed fix is negligible, however, it is
prioritized for a future update.
```

## 6.9 Unchecked Return Values and Non-Compliant `IGem` Interface

Design | Low | Version 1 | Risk Accepted

The `IGem` Interface is used when handling collateral inside the MultiplyProxyActions contract. The return values of these calls is, however, never checked. While most ERC-20 tokens revert on a failed transfer, according to the ERC-20 specification it is sufficient to return `false`. A collateral with this behavior may not be supported correctly.

```
function approve(address, uint) virtual public;
function transfer(address, uint) virtual public returns (bool);
function transferFrom(address, address, uint) virtual public returns (bool);
```

Additionally, note that the function `approve` defined in `IGem` has no return value. According to the ERC-20 specification ( https://eips.ethereum.org/EIPS/eip-20 ) this function must have a boolean return value:

```
function approve(address _spender, uint256 _value) public returns (bool success)
```

---

**Risk accepted:**

Oazo Apps Limited replied:

```
Such implementation is compliant with the current Maker Protocol - it was consulted
and confirmed with the Maker DAO’s Protocol Engineering Core Unit.
```

## 6.10 Unnecessary Call of `getAaveLendingPool`

Design | Low | Version 1 | Acknowledged

After receiving the requested flash loan from Aave, its lending pool contract calls function `executeOperation`. As the lending pool is the message sender of this call, it would be unnecessarily expensive to retrieve the lending pool address from Aave's lending pool provider. However, in `executeOperation` of contract MultiplyProxyActions the lending pool is retrieved as follows:

```
ILendingPoolV2 lendingPool = getAaveLendingPool(addressRegistry.aaveLendingPoolProvider);
```

`getAaveLendingPool` retrieves the Aave lending pool address from the lending pool provider. Thus, gas could be saved.

---

**Acknowledged:**

Oazo Apps Limited replied:

```
Optimisation will be done in a future version of the contract.
```

## 6.11 Variables Could Be Immutable and Constant

Design | Low | Version 1 | Code Partially Corrected

In Exchange.sol the token swap logic is defined. The state variable `feeBase` is initialized upon declaration and the state variable `fee` is initialized in the constructor. Both cannot be modified. Hence, they could be declared as constant and immutable respectively.

---

**Code partially corrected:**

`feeBase` was made `constant`. `fee` now has a setter and can no longer be immutable. However, with the updates Oazo Apps Limited decided to stick with the `feeBeneficiaryAddress` in Exchange.sol. However, this variable cannot be modified and, hence, it could be made immutable.

## 6.12 `_getWipeDart()` Returns `art` When Closing

Design | Low | Version 1 | Acknowledged

`_getWipeDart()` is used in `MultiplyProxyActions.wipeAndFreeGem()` to determine the amount for `dart` to be passed to `frob()`. For closing operations, the intention is to wipe the whole art of the urn and, thus, calling `_getWipeDart()` creates an overhead in computation as it will return the whole art of the urn. The actual value would be available even before calling `wipeAndFreeGem()`.

When closing a vault, immediately before the call to `wipeAndFreeGem()` the `ink` of the urn is querried from the `vat`:

```
(uint256 ink, ) = IVat(vat).urns(cdpData.ilk, urn);
```

Note that the second return value, the art of the urn is dropped. Hence the value would be available without any meaningful extra gas overhead. Therefore, the cases of wiping some and wiping all art when freeing gem could be distinguished, as it is similarly done in the DSS Proxy Actions contract, to reduce the gas cost of multiply closing actions.

---

**Acknowledged:**

Client responded that they want to remain consistent with the original Proxy Actions in the Maker Protocol and refers to the DssProxyActions contract.

In the function of DssProxyActions refered to, `wipeAllAndFreeETH` is public and works with the arguments passed. In MultiplyProxyActions, `wipeAndFreeGem` is an internal function. Depending on the call path one already knows whether it's a full or partial wipe. Notably in the case of a full wipe (closure of the vault) one would already know the amount for `dart` due to the call to `(uint256 ink, ) = IVat(vat).urns(cdpData.ilk, urn);` (where the returned value for `art` is currently ignored). Note that the original Proxy Actions contract implements two different functions `wipeAllAndFreeGem()` and `wipeAndFreeGem()`. In the original Proxy Actions, `wipeAllAndFreeGem()` queries `vat.urns()` for `dart` while in MultiplyProxyActions this call is already made in the function calling `wipeAndFreeGem()` so the identical functions cannot but the concept could be reused.

## 6.13 `transfer` Is Used for ETH Transfers

Design | Low | Version 1 | Risk Accepted

Solidity offers different options to perform ETH transfers. When performing a decrease action withdrawing collateral, MultiplyProxyActions receives ETH in return for the WETH withdrawn from the vault in function `_withdrawGem()`. In the same function, the solidity `transfer` feature for sending ETH from the MultiplyProxyActions contract to the user. As `transfer` sends a fixed gas cost along with the funds, it is assumed that EVM gas costs remain constant in the future. Note that gas costs have change in the past, which has led to issues with the use of `transfer`. Hence, possible integration consequences need to be considered.

---

**Risk accepted:**

Oazo Apps Limited replied:

```
Malfunction requires Ethereum hard fork. We will update the
contract if that occurs.
```

# 7 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 | 7 |

- withdrawCollateral Is Added to borrowCollateral Specification Changed
- Disabled Optimizer Code Corrected
- Impossible Decrease Operations Code Corrected
- Incorrect Decimals When Handling Collateral Code Corrected
- Unclear Parameter Specification Specification Changed
- Undocumented Public Functions Code Corrected
- feeRecepient of AddressRegistry Code Corrected

| | |
|---|---|
| Low-Severity Findings | 10 |

- Cannot Update Fees Code Corrected
- Specification Mismatches Specification Changed
- Unused Constant Code Corrected
- skipFL Case in Flashloan Callback Code Corrected
- Code Duplication Code Corrected
- No Check on Amount Received From Flash Loan Code Corrected
- Possible Casting Overflow Code Corrected
- Unnecessary transferFrom Code Corrected
- Use Available Constant and Avoid Call Code Corrected
- increaseMultipleDepositDai Is Payable Code Corrected

## 7.1 `withdrawCollateral` Is Added to `borrowCollateral`

Correctness Medium Version 2 Specification Changed

In the updated specification received after the intermediate report, `borrowCollateral` is specified as follows:

```
If a Multiply decrease action: the amount of collateral that
is needed to decrease multiple (it includes the ``withdrawCollateral`` amount if any is
specified).
```

That implies that `borrowCollateral` already includes `withdrawCollateral`. However, in function `_decreaseMP` the amount of collateral to draw from the vault passed to `wipeAndFreeGem()` is computed as follows:

```
cdpData.borrowCollateral.add(cdpData.withdrawCollateral)
```

Thus, `withdrawCollateral` is accounted twice and `wipeAndFreeGem()` will exit more collateral than intended.

---

**Specification changed:**

The documentation has been updated and no longer states that `borrowCollateral` includes `withdrawCollateral`.

## 7.2 Disabled Optimizer

Design | Medium | Version 1 | Code Corrected

The solidity optimizer has been disabled in the hardhat configuration:

```
solidity: "0.7.6",
    settings: {
        optimizer: {
            enabled: false,
            runs: 1000
     }
},
```

The optimizer reduces both code size (thereby deployment costs), and execution costs.

---

**Code corrected:**

The optimizer was enabled.

## 7.3 Impossible Decrease Operations

Design | Medium | Version 1 | Code Corrected

`wipeAndFreeGem()` is used in each decrease operation to withdraw collateral from the vault to the MultiplyProxyActions contract. However, the function reverts if the collateral has less than 18 decimals.

```
function wipeAndFreeGem(
    address manager,
    address gemJoin,
    uint256 cdp,
    uint256 borrowedDai,
    uint256 collateralDraw
) public {
    ...
    uint256 wadC = convertTo18(gemJoin, collateralDraw);
    IManager(manager).frob(cdp, -int256(wadC), _getWipeDart(vat, IVat(vat).dai(urn), urn, ilk));
    IManager(manager).flux(cdp, address(this), wadC);
    IJoin(gemJoin).exit(address(this), wadC);
}
```

Following scenario could occur if the collateral is GUSD which has only two decimals.

1. Parameter collateralDraw is converted to 18 decimals representations which is stored in local variable wadC. Meaning that wadC == collateralDraw * (10**16).
2. frob and flux are called with wadC as part of the argument.
3. GemJoin.exit() is also called with wadC as an argument.
4. In GemJoin contract's exit(), the GemJoin contract will try to call the GUSD contract to transfer wadC tokens.
5. The transaction reverts since wadC is much higher than the balance in GUSD of GemJoin at that moment.

Since exiting creates a transfer in the GUSD contract, it must use the amount of decimals the token has.

---

**Code corrected:**

collateralDraw instead of wadC is now passed to the call to gemJoin.exit() which is in the correct unit.

## 7.4 Incorrect Decimals When Handling Collateral

Correctness Medium Version 1 Code Corrected

The functions _closeWithdrawCollateralSkipFL and _closeWithdrawCollateral receive multiple inputs including the ink of the relevant urn. The ink value has previously been queried from the vat. The ink value is then used as follows:

```
require(
  IERC20(exchangeData.fromTokenAddress).approve(address(exchange), ink),
  "MPA / Could not approve Exchange for Token"
);
```

The ink value, however, has been adjusted to 18 decimals and hence will be incorrect here for all tokens that do not have 18 decimals.

Note that for the functions _closeWithdrawCollateral and _closeWithdrawDai the ink value is also incorrectly passed to wipeAndFeeGem.

---

**Code corrected:**

The updated implementation now passes cdpData.borrowCollateral instead of ink when calling _closeWithdrawCollateralSkipFL and _closeWithdrawCollateral. Inside the called function this parameter is called ink. Althrough this solution technically works, it is not ideal:

- Note that both functions already take the struct cdpData as parameter, so passing cdpData.borrowCollateral separately is redundant.
- The call to token.approve() remains unchanged. The exchange is approved to transfer the amount ink (which now is cdpData.borrowCollateral) however the amount the exchange will transfer is exchange.fromTokenAmount.

**Risk accepted:**

Refactoring / improvements are planned after the MVP.

An additional problem for collaterals with non 18 decimals has been uncovered in `_closeWithdrawCollateralSkipFL()` after the draft report: `wipeAndFreeGem()` expects the collateral amount in the unit of the collateral token and converts it to the 18 decimal representation. However in `_closeWithdrawCollateralSkipFL` this conversion is already done before the second call to `wipeAndFreeGem()`, hence the conversion will happen twice and result in an incorrect value for collaterals with less than 18 decimals. This has been correct by removing the conversion before the call to `wipeAndFreeGem()`.

## 7.5 Unclear Parameter Specification

Design Medium Version 1 Specification Changed

There are several input parameters to each call. These are passed in three different structs: `ExchangeData`, `CdpData`, `AddressRegistry`. Some definitions of the parameters of the strucs are unclear and may lead to mistakes.

- `borrowCollateral` is specified to be the collateral to buy with the flashloan in increase operations. However, it is used differently. More precisely, it is used in function `_decreaseMP` to specify how much collateral to be withdrawn from the vault. The frontend may create mistakes. If the intention of `borrowCollateral` is to be used as the amount that will be, together with `depositCollateral`, deposited to the vault, then with positive slippage, `joinDrawDebt` would deposit too much collateral into the vault.
- `depositCollateral` is specified to be the amount of collateral the user deposits in increase actions that deposit collateral. For ETH that is not the case since `msg.value` is used and no check if it equals `depositCollateral` is done. The call will work but the result may differ from the expected behaviour.
- `fromTokenAmount` is specified to be the amount of tokens to be exchanged. `depositDai` is the amount of `DAI` that should be exchanged jointly with the flashloaned DAI in increase operations. In `_increaseMP` in the call to swap the tokens to collateral, the to be swapped amount is specified as the sum of `fromTokenAmount` and `depositDAI`. However, according to specification, `fromTokenAmount` should already be accounting for the deposit.
- Depending on what the intended use of the parameters is, specifying invariants could be helpful to clarify for example whether the `fromTokenAmount` in increase operations is the sum of the flashloaned and deposited `DAI`.

Clarifying these and similar ambiguities may help users to understand the parameters of their transactions better.

---

**Specification changed:**

The parameters are now more precisely defined.

## 7.6 Undocumented Public Functions

Design Medium Version 1 Code Corrected

Function `wipeAndFreeGem` of the `MultiplyProxyActions` contract is `public`. The function is only used internally and there is no valid use case to call it directly. Direct calls to this function will likely fail due to preconditions not being met. Furthermore the documentation does not list it as one of the public functions.

Similarly `_collectFee()` of the Exchange contract is `public` despite being used only internally. Here as well the documentation does not list `_collectFee()` as public function.

**Code corrected:**

The functions are now `internal`.

## 7.7 `feeRecepient` of `AddressRegistry`

Correctness | Medium | Version 1 | Code Corrected

In the last paragraph of section `Architectural decisions` of the specification document it's mentioned that the `feeRecipient` is part of the off-chain registry (the struct `AddressRegistry` passed as function parameter):

> A caveat that has been raised is that using an off-chain registry managed by the frontend opens for other frontends using our smart contract while passing their own fee wallet address in the params.

This struct features a field `feeRecepient`:

```
struct AddressRegistry {
    address jug;
    address manager;
    address multiplyProxyActions;
    address aaveLendingPoolProvider;
    address feeRecepient;
    address exchange;
}
```

However in the smart contracts reviewed, this field is never read. The exchange contract uses a `feeBeneficiary` variable set in the constructor.

---

**Code corrected:**

`feeRecepient` was removed from the struct.

## 7.8 Cannot Update Fees

Design | Low | Version 2 | Code Corrected

In version two, `setFee()` was introduced to enable updating the `fee` variable for fees going to Oazo. The caller must be authorized. However, the only authorized caller is the MultiplyProxyActions contract which does not call `setFee()` and, hence, the fees cannot be update.

---

**Code corrected:**

The `feeBeneficiary` is now also whitelisted and can set fees.

## 7.9 Specification Mismatches

Design | Low | Version 2 | Specification Changed

In version two, new parameters to the `CdpData` struct were introduced. However, the `methodName` element in the struct is unspecified in the documentation while it is used in the code.

---

**Specification changed:**

`methodName` was added to the documentation.

## 7.10 Unused Constant

Design | Low | Version 2 | Code Corrected

In MultiplyProxyActions, a constant ETH_ADDR is defined. However, it is not used in the code.

---

**Code corrected:**

The constant was removed.

## 7.11 `skipFL` Case in Flashloan Callback

Design | Low | Version 2 | Code Corrected

Function `executeOperation` is the callback for the Aave flashloan. It is not intended to be used if flashloans are not utilized.

The following code snipped can be found at the end of the function:

```
if (cdpData.skipFL == false) {
  IERC20(assets[0]).approve(
    address(getAaveLendingPool(addressRegistry.aaveLendingPoolProvider)),
    borrowedDaiAmount
  );
}
```

---

**Code corrected:**

The code was removed.

## 7.12 Code Duplication

Design | Low | Version 1 | Code Corrected

When crafting the call to and calling the flashloan contract, multiple functions of the `MultiplyProxyActions` contract have the same code. This code duplication could be removed by moving the respective code into an internal function.

---

**Code corrected:**

The common parts have been extracted into a function `takeAFlashLoan`.

## 7.13 No Check on Amount Received From Flash Loan

Correctness Low Version 1 Code Corrected

The documentation specifies following check on the amount received from the flash loan:

> We check in our smart contract whether the delivered amount matches our expected amounts. If not enough funds are delivered we revert the transaction with the message: “FL malfunction”.

However, this check is not made. Moreover, that could allow a malicious Aave to send less funds than requested. Performing an increase operation with personal `DAI` besides the flashloan may lead to Aave being paid more if the front-end does not set parameters such that this transaction would revert. In any case, there is a mismatch between documentation and implementation.

---

A require statement has been added to check whether sufficient funds have been received:

```
require(
  cdpData.requiredDebt == IERC20(DAI).balanceOf(address(this)),
  "requested and received amounts mismatch"
);
```

However note that this require statement now breaks `increaseMultipleDepositDai()` for the case when a flashloan is used: The DAI amount to deposit has already been transferred onwards to `MultiplyActionsProxy`, this amount will be included in the in the DAI balance and hence the balance will not equal the flashlaon amount. The transaction will revert.

**Code corrected:**

The check has been changed to:

```
require(
  cdpData.requiredDebt.add(cdpData.depositDai) <= IERC20(DAI).balanceOf(address(this)),
  "requested and received amounts mismatch"
);
```

For the final version of the code the strict requirement for equality has been weakened to allow for surplus DAI balance at the contract.

## 7.14 Possible Casting Overflow

Design Low Version 1 Code Corrected

In function `wipeAndFreeGem` it is necessary to cast the amount of collateral to be withdrawn from `uint256` to `int256` so it can be passed as an argument when calling `frob()` on the manager. The cast can overflow since it is using a regular cast and, hence, `frob()` could be called with wrong parameters. Instead, `toInt256()` could be used since it reverts if overflows occur.

---

**Code corrected:**

`toInt256()` is now always used for casting from `uint256` to `int256`.

## 7.15 Unnecessary `transferFrom`

Design Low Version 1 Code Corrected

In function `wipeAndFreeGem` the MultiProxyActions contracts transfers `DAI` from itself to itself.

```
IDaiJoin(DAIJOIN).dai().transferFrom(address(this), address(this), borrowedDai);
```

As this call does not modify any `DAI` balance, it could be removed to save gas.

---

**Code corrected:**

The call was removed.

## 7.16 Use Available Constant and Avoid Call

Design Low Version 1 Code Corrected

In `MultiplyProxyActions.wipeAndFreeGem()` `DAI` is handled as follows:

```
IDaiJoin(DAIJOIN).dai().transferFrom(address(this), address(this), borrowedDai);
IDaiJoin(DAIJOIN).dai().approve(DAIJOIN, borrowedDai);
```

Instead of using the `DAIJOIN` address and calling it repeatedly to query the `DAI` address, the constant `DAI` representing the address of the `DAI` contract could be used directly.

---

**Code corrected:**

The constant `DAI` is now used directly.

## 7.17 `increaseMultipleDepositDai` Is Payable

Design Low Version 1 Code Corrected

The system allows users to deposit additional DAI that will be, along with the flash-loaned DAI, swapped to the underlying collateral to enable increasing the leverage position. This functionality is implemented in the `payable` function `increaseMultipleDepositDai`. However, this method does not require having any additional ETH collateral and, thus, ETH could be mistakenly transferred to the contract executing the `delegatecall` to `MuliplyProxyActions`.

---

**Code corrected:**

In the updated code `increaseMultipleDepositDai` is no longer `payable`.

## 7.18 `transferFrom` Used Instead of `transfer`

Note Version 1 Code Corrected

In function `_collectFee` of Exchange.sol, `IERC20(asset).transferFrom(address(this), feBeneficiaryAddress, feeToTransfer)` could be simplified to

`IERC20(asset).transfer(feeBeneficiaryAddress, feeToTransfer)`. `transfer()` could be safer to use if, at any point in the future, Oazo decides to accept other tokens than `DAI` for fees, even though this is currently not the case nor is it planned for the future. For example, that call would revert if `asset` was `USDT` which reverts if the contract has not approved itself in such a scenario.

---

**Code corrected:**

The code was changed to use `safeTransfer()`.

# 8 Notes

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

## 8.1 Deprecated `safeApprove`

**Note** **Version 1**

In function swap of the Exchange, `safeApprove()` is called. However, a deprecated implementation of `safeApprove()` is used. Be aware that using the most recent implementation, `safeApprove()` may revert if the approval has not been set to zero.

## 8.2 Documentation Treats Should as Must

**Note** **Version 1**

The documentation uses *should* when describing what values the parameters passed from the frontend to the backend will have.

RFC 2119 defines *should* as follows:

> *SHOULD* This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

For example, following exchange parameter is defined as follows in the documentation:

> `allowPartialFill` - Value should be set to false in order to ensure that the whole amount is swapped, otherwise the tx will fail.

That allows `allowPartialFill` to be `true` in certain scenarios. However, it could be dangerous to set this value to `true` since in `_closeWithdrawDai()` the contract swaps collateral for DAI and does not check whether there is some unswapped collateral returned from the exchange. Hence, users could lose funds. Clarifying how parameters are set clearly may help avoid mistakes.

## 8.3 Gas Costs Vs Flashloan Fee

**Note** **Version 1**

`MultiplyProxyActions` uses a flashloan to leverage the exposure of the collateral in one action. There is an implicit assumption that the fee for the flashloan is less than what the gas costs would be to reach the same multiply ratio by using the borrowed dai to buy collateral and to borrow additional dai repeatedly. For very large amounts the flashloan fee may exceed the gas cost and using the `MultiplyProxyActions` may be more expensive than doing the actual steps repeatedly.

---

Oazo Apps Limited replied:

```
From our experience in implementing a Multiply solution using repeated actions on Vaults
we do not believe this approach to be feasible from a practical point of view. Further,
going forward we expect to integrate to several flash loan providers and we expect the
flash loan fee to go towards zero.
```

## 8.4 No Checks on Outcome

**Note** **Version 1**

The `MultiplyProxyAction` smart contract does not check the actual outcome of an action on a vault. However, the execution highly depends on the front-end and the APIs it interacts with and has much interaction with external contracts. In such systems, there is typically an optional feature enabling users to enforce certain postconditions such as a minimum collateralization ratio or other similar properties in order to safeguard their actions.

## 8.5 Opposite Effect of What the Function Name Suggests Possible

**Note** **Version 1**

Note that while in general `increaseMultiple...` increases and `decreaseMultiple...` decreases the leverage of a vault, due to the added possibility of adding or withdrawing within the same action the final collateralization ratio / leverage may be changed in the opposite direction to what the function name increase/decrease suggested.

The documentation does not describe this behavior in detail.

## 8.6 Whitelisted Contracts for Exchange Not Public

**Note** **Version 1**

In Exchange.sol, there is an internal mapping `WHITELISTED_CALLERS` specifying which contracts can use the Exchange contract. Since currently only the MultiplyProxyActions will be set in the constructor, it is relatively simple to decide whether a contract is whitelisted or not. However, in the future more contracts may be whitelisted. Thus, making this mapping public could be useful.

## 8.7 `CDPManager` Must Have Created Vault

**Note** **Version 1**

The `MultiplyProxyActions` contract is only compatible with vaults that have been opened through the `CDPManager`. Vaults that have been opened directly are not supported as the `CDPManager` is unable to acquire permission to modify the vault. The documentation is vague on this topic and may be clarified.

## 8.8 `closeVaultExitCollateral` Surplus `DAI` Transferred to `fundReceiver`

**Note** **Version 1**

Function `closeVaultExitCollateral` of the `MultiplyProxyActions` contracts can be used to close a vault and exit all funds in collateral to the `fundReceiver` address. A part of the collateral is necessary to repay the flashloan and hence is exchanged into `DAI`. Note that should there be surplus

DAI tokens after the exchange, these are transferred to the fundReceiver address in addition to the collateral.

## 8.9 msg.value Unchecked for ERC-20 Deposits

**Note** **Version 1**

In function increaseMultipleDepositCollateral, collateral is deposited. msg.value should be non-zero if the collateral to lock in the vault will be WETH. However, it could also be non-zero if the collateral is a ERC-20 token. The ETH sent along with the ERC-20 will be unused will remain in the contract after the transaction has finished.