PUBLIC

Code Assessment

of the Modular Proxy Actions Smart Contracts

September 14, 2022

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 9
4 Terminology 10
5 Findings 11
6 Resolved Findings 13
7 Notes 24

1 Executive Summary

Dear all,

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

The modular proxy actions allow execution of operations, a set of actions. An action contract performs a single function. This flexibility makes it trivial to compose new operations from actions, especially as actions may be added or upgraded.

The most critical subjects covered in our audit are functional correctness, security and whether the implementation is suitable for the intended purpose.

While the modular implementation is suitable to reach the documented requirements it results in increased transaction costs which may hinder adoption. The modularity is significantly more complicated than a monolithic architecture. Extensive forked mainnet tests are recommended.

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	2
• Code Corrected	2
Medium-Severity Findings	9
• Code Corrected	8
• Specification Changed	1
Low-Severity Findings	14
• Code Corrected	11
• Specification Changed	1
• Risk Accepted	2

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 Modular Proxy Actions 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	18 July 2022	755ee7c4aa3f27fb99c742addfe51c9303ffc415	Initial Version
2	10 August 2022	f18bd884079b3532e6a3567577dc216bba3ff9e2	After Intermediate Report
3	30 August 2022	70cc810526353e08b75575b571271425c99b54b6	Updated Version
4	12 September 2022	5ab9875657263d75d96d75b33e444438e015cb2d	Final Version

For the solidity smart contracts, the compiler version `0.8.5` was chosen. After the intermediate report the compiler version was updated to `0.8.15`.

The following files are in scope of this review:

- contracts/actions/aave:
    - Borrow.sol
    - Deposit.sol
    - Withdraw.sol
- contracts/actions/maker:
    - CdpAllow.sol
    - Deposit.sol
    - Generate.sol
    - OpenVault.sol
- contracts/actions/common:
    - Executable.sol
    - PullToken.sol
    - SendToken.sol
    - SetApproval.sol
    - TakeFlashloan.sol
    - UseStore.sol

- contracts/core/ except ServiceRegistry.sol and McdView.sol. Earlier versions of these contracts have been reviewed as part of another report.

2.1.1 Excluded from scope

All files not listed as in scope above, including:

- interfaces/*
- libs/* (excluding libs/DS/ProxyPermission.sol which is in scope)
- test/*

Notably actions maker/Payback.sol, maker/Withdraw.sol and all actions which were not present in the initial commit of this review despite being part of the final commit listed have not been reviewed.

2.2 System Overview

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

The smart contracts of the modular proxy actions allow to execute a series of calls aggregating actions. Individual actions may use return values or data stored by previous actions. A special kind of action allows taking a flashloan and continuing execution of actions thereafter.

Actions are individual smart contracts. Their code is executed using Delegatecall in the current context of the execution. All contract addresses, external and internal are queried from the ServiceRegistry. This concept allows updating or adding new actions seamlessly.

Currently the following actions are implemented:

- **PullToken:** Pulls tokens of the specified asset from the given source address to the current executing account (`address(this())`. Requires prior approval for the token transfer to succeed.
- **SendToken:** Sends an amount of tokens or Ether to the given address.
- **SetApproval:** Gives approval to transfer an amount of an asset to the given address.
- **SwapOnOneInch:** Calls the 1inch aggregator to execute a swap. Gives approval before and ascertains that the balance after exceeds the minimum amount specified.
- **TakeFlashloan:** Takes a flashloan. If `dsProxyFlashloan` is specified, gives permission to the OperationRunner to call `Execute(address,bytes)` on the DsProxy. This permission is revoked afterwards OperationExecutor implements the required `onFlashloan()` interface.

For Maker:

Interactions happen on vaults managed by the CdpManager:

- **OpenVault:** Opens a new vault. The `ilk` type is defined by the specified join adapter. Pushes the id to the OperationStorage.
- **CdpAllow:** Give allowance for a vault in the CdpManager to the specified address. The CdpID may be retrieved from OperationStorage.
- **Deposit:** Deposit collateral (token/Ether) from the current executing account as `gem`, immediately lock it as `ink`. The CdpID may be retrieved from OperationStorage.
- **Generate:** Makes the specified amount of DAI tokens available. If required, generates more debt (`art`) in the urn. The CdpID may be retrieved from OperationStorage. The DAI amount is pushed to the OperationStorage.

For Aave v2:

- **Deposit:** Deposits the specified amount into the lending pool. Receives aTokens in exchange. Pushes the amount to OperationStorage.
- **Withdraw:** Withdraws the specified amount by redeeming aTokens.
- **Borrow:** First approves the delegation before borrowing ETH. The receiver must be able to receive Ether. Currently the OperationExecutor does not support this.

Actions may be combined as desired in so-called operations. The following example demonstrates this capability:

IncreaseMultipleWithFl, which in the provided test case consists of the following actions: OpenVault, PullToken, Deposit, TakeAFlashloan, Swap, Deposit, Generate, sendToken.

The setup allows support of arbitrary DeFi systems, extending the capabilities by adding new actions or by crafting new operations by aggregating actions is intended to be simple.

OperationExecutor

The main contract is the OperationExecutor. It features the following entry points:

- `executeOp()`: The user will execute this function through his DsProxy as delegatecall. Initializes/resets OperationStorage.
- `aggregate()`: Executes calls, executes the code of the individual actions as delegatecall. Although the function is public, it should not be called directly.
- `onFlashloan()`: Implements the interface for the callback of the ERC3156 compliant flash loan provider. If the flag `dsflashloan` is true, continues execution of the following action via the DsProxy, otherwise calls `aggregate()` directly.

The AutomationBot may uses the OperationExecutor. It executes `executeOp()` via Delegatecall similarly to the DsProxy. For Flashloans, `dsProxyFlashloan` will have to be set to `false`. The callback to `onFlashLoan` will the execute the following actions directly in the context of OperationExecutor. In this case the AutomationBot will have to give allowance to the OperationExecutor to operate on this CdpID in the CdpManager first. It must be made sure the no funds remain in / controlled by the OperationExecutor and that the allowance on the CdpID is revoked again upon completion of the execution.

OperationsStorage

Used as temporary storage, assumed to be empty at the beginning of every execution. Actions can push to and read from an array of bytes32. Function `execute()` specified in Interface Executable takes a paramsMap array as argument. Non zero values mean replacing this argument by the value at position x-1 in the operations storage array.

Additionally, implements the logic to verify actions. Verifying actions checks whether the calls passed to `executeOp` matches the call sequence stored for this operation in the OperationRegistry. No check is done when OperationRegistry returns an empty set of calls for a given operation name.

OperationsRegistry

This contract keeps a mapping of operations identified by a string. An operation is a array of bytes32.

The mapping can be updated (currently in Version 1 without access control) and queried.

2.3 Trust Model & Roles

Front End: Fully trusted to craft a correct set of calls including proper calldata.

Oasis: Fully trusted to operate the OperationsRegistry correctly and to add genuine and correct actions executing as documented only.

AutomationBot: Fully trusted.

User: Untrusted, any user may interact with the contracts through the context of their own DsProxy. 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).

Users may also interact with the operation executor directly. This is not an intended use case. As the contract is stateless outside of operations, this should not have an impact.

CdpManager: Trusted, the CdpManager contract of Maker.

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.

All tokens are assumed to be ERC20 compliant without special behavior. No supported token can have more than 18 decimals.

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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	0
Low-Severity Findings	2

- FeeTier of Swap Risk Accepted
- Missing NatSpec Risk Accepted

5.1 FeeTier of Swap

Design Low Version 1 Risk Accepted

The Swap contract allows authorized callers to add multiple fee tiers. These tiers are not documented. Any fee added as fee tier is valid. All calls to `swapTokens()` may simply specify the lowest allowed fee, higher fee tiers can just be avoided by the users.

Risk accepted:

Oazo Apps Limited states:

```
Added a note at the bottom of the Operation Registry section. We accept the risk
that the user might change the fee from the front-end.
```

5.2 Missing NatSpec

Design Low Version 1 Risk Accepted

The code is not documented in the NatSpec format. It is recommended to fully annotate all public interfaces. This should help both end users and developers interact with the contracts.

Risk accepted:

To be added later.

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	2

- OperationStorage Can Be Polluted Code Corrected
- OperationsRegistry: No Access Control Code Corrected

Medium-Severity Findings	9

- Implementation of UseStore Inconsistent With Documentation Specification Changed
- Ineffective receiveAtLeast Check After Swap Code Corrected
- Maker Deposit Action Uses Full Balance Code Corrected
- No Access Control on onFlashLoan Code Corrected
- OperationRegistry: No Entry, No Checks Code Corrected
- Payable Action.execute() Code Corrected
- Reentrancy Into executeOp() Code Corrected
- Visibility of aggregate Function Code Corrected
- sendToken: Transfers msg.value Instead of send.amount Code Corrected

Low-Severity Findings	12

- Action Events Not Emitted Code Corrected
- Actions: Inconsistent Destination of Tokens Code Corrected
- DAI Address Could Be Constant Code Corrected
- OperationStorage: Unused owner Variable Code Corrected
- OperationsRegistry: No Events Emitted on State Change Code Corrected
- Outdated Compiler Version Code Corrected
- Receiver of Flashloan Code Corrected
- Sanity Check in on Flashloan Code Corrected
- Swap Slippage Saved Event Order Code Corrected
- Swap.sol: ReceiveAtLeast Does Not Take Into Account the Fee Specification Changed
- Unused Return Value of Aave Withdraw Code Corrected
- onFlashLoan() Ignoring Fees Code Corrected

6.1 OperationStorage Can Be Polluted

Security High Version 1 Code Corrected

OperationStorage is designed to be used as a temporary store of actions and return values for the execution of an operation. Therefore, the variables it contains are deleted at the end of every operation execution. However, as it lacks access control, and since there is no mechanism to ensure that OperationStorage is empty before an execution, action and return values could be maliciously or erroneously introduced.

In particular, an attacker could store spurious return values with the `push` function. In the next execution actions may retrieve these values instead of the intended ones which are appended at the end of the array.

Finally, it should be considered that the execution of actions may reach untrusted code (integrations, tokens). Functions `push` and `finalize` may be accessed unexpectedly even within the execution of an action. This similarly applies to functions `setOperationActions` and `verifyAction` were it is not obvious whether this can have a negative impact.

Code partially corrected:

OperationStorage is now cleared at the beginning of `OperationExecutor.executeOp()`, this ensures that the execution of operation does not start with a polluted OperationStorage which mitigates the main issue.

Within execution of actions untrusted code may be reached (integrations, tokens), in theory they may execute state changing functions of the OperationStorage: `push()`, `verifyAction()`, `clearStorageAfter()`.

Code corrected:

OperationStorage contract now stores the return values from actions in a mapping where values are assigned to the address that pushed them.

```
mapping(address => bytes32[]) public returnValues;

function push(bytes32 value) external {
...
returnValues[msg.sender].push(value);
}

function at(uint256 index, address who) external view returns (bytes32) {
return returnValues[who][index];
}
```

When writing to the OperationStorage, an address can only write in the array associated to this address. When reading from the OperationStorage, the caller must specify which value from which address he wants to read. This prevents untrusted code to tamper with the return values during an operation.

In case of a flashloan action executed from the AutomationBot (more precicesly a Flashloan action with flag `dsProxyFlashloan` set to `false`) execution continues in the context of the OperationExecutor. The original initiator will be pushed to the OperationStorage, when called from the OperationExecutor functions `push`, `at` and `len` will use this address instead of `msg.sender`.

6.2 OperationsRegistry: No Access Control

Security | High | Version 1 | Code Corrected

The purpose of the OperationsRegistry contract is to specify the set of actions identified by a `string name`. As this contract lacks access control, anyone could modify the mapping between operation and actions. This can be done through the `addOperation` function, which allows not only adding a new operation but also modifying any existing one.

As a consequence, an attacker could modify the entries for existing operations. This could prevent the corresponding verifications from succeeding, and thus compromise the availability of the system. An attacker may as well delete the actions stored for an operation resulting in no verification on the calls being done in `OperationExecutor.aggregate()`.

The extensive documentation lacks a description of the OperationRegistry.

Code corrected:

Access control has been added: There is now an `owner`, only this `owner` can add/update operations to the OperationRegistry.

6.3 Implementation of UseStore Inconsistent With Documentation

Correctness **Medium** **Version 1** **Specification Changed**

The implementation of the `read` function in `UseStore` is not consistent with the provided documentation. The function shown in the PDF does not subtract 1. In general, mixing 0-based and 1-based indexing can be a source of errors, so this should be well documented.

Specification changed:

The excerpt of `UseStore.read()` shown in the documentation has been updated and is now in line with the actual implementation. Furthermore the params mapping section of the documentation has been extended.

6.4 Ineffective `receiveAtLeast` Check After Swap

Correctness **Medium** **Version 1** **Code Corrected**

To verify that the swap executed correctly, `SwapOnOneInch.execute()` checks that the balance is at least what the user wanted to receive:

```
require(balance >= swap.receiveAtLeast, "Exchange / Received less");
```

The check doesn't take into account that the token balance before the swap may have been non-zero already. Hence the check may pass despite the swap resulted in less than `receiveAtLeast` tokens. This code is intended to be executed as Delegatecall in the context of the users DsProxy, a non-zero balance of the token out before the swap is not an unlikely scenario.

A similar check is done in `Swap.sol`. This contract however is used differently: It's a helper contract which is not supposed to hold any token balances in between calls, furthermore it forwards all token balance. Hence in the Swap contract; from a caller's perspective the check ensures `receiveAtLeast`.

Code corrected:

File SwapOnOneInch.sol no longer exists in the updated codebase.

6.5 Maker Deposit Action Uses Full Balance

Correctness Medium Version 1 Code Corrected

While the `DepositData` struct contains an `amount` parameter, the `maker/Deposit` action always uses the full available balance. This behavior is not documented and may be unexpected for users who specify an inferior amount.

Code partially corrected:

The code of action maker/Deposit now deposits the amount specified. However the action still exchanges all Ether balance to WETH. Is this intended?

Code corrected:

The code wrapping ETH has been removed. This fixes the remaining issue as the user's Ether will not be exchanged to WETH. Note that the user needs to have WETH available instead.

6.6 No Access Control on `onFlashLoan`

Design Medium Version 1 Code Corrected

The `onFlashLoan` function of the `OperationExecutor` contract is only intended to be called by the Flashloan provider. As it has no access control it can be called by anyone. The contract will then give approval to the registered lender for `amount` of `asset`. While this is not necessarily a problem, it breaks the normal pattern that the OperationExecutor is "stateless" in between calls, in the sense that he has given an approval to transfer tokens to a third party.

Code corrected:

Access control has been added to `OperationExecutor.onFlashLoan()`. The function can only be called by the trusted lender returned by the Registry:

```
address lender = registry.getRegisteredService(FLASH_MINT_MODULE);
require(msg.sender == lender, "Untrusted flashloan lender");
```

6.7 OperationRegistry: No Entry, No Checks

Design Medium Version 1 Code Corrected

When there is no operation stored for a name, `getOperation()` returns an empty array and subsequently nothing is checked. Shouldn't this case be handled explicitly to avoid not checking correctness by accident?

An operation name is a string. This allows displaying the operation name in a human readable way. However, this can be dangerous as strings support the Unicode charset and many lookalike characters of different alphabets exist in this charset. Hence users might be tricked.

For more insights into lookalike characters, please refer to: https://util.unicode.org/UnicodeJsps/confusables.jsp?a=IncreaseMultipleWithFl

---

Code corrected:

`getOperation()` of OperationRegistry now reverts on non-existing operations instead of returning an empty array (which results in skipping checks). Custom operation with empty actions have to be explicitly added to the OperationRegistry.

6.8 Payable `Action.execute()`

Design | Medium | Version 1 | Code Corrected

The interface `Executable` specifies:

```
function execute(bytes calldata data, uint8[] memory paramsMap) external payable;
```

The code of actions is executed as delegatecall from within `OperationExecutor.aggregate()`. Delegatecall preserves `msg.sender` and `msg.value`. The aggregate function of the OperationExecutor is not payable, hence `msg.value` will always be zero. Calls to `executeOp()` / `aggregate()` with non-zero `msg.value` will revert, hence why is the `execute` function of actions supposed to be payable?

Note that actions may still work with Ether despite not receiving calls with non-zero `msg.value`: Ether can be received by the DsProxies fallback function / the DsProxy can already have an Ether balance which can be transferred onwards.

Code corrected:

`OperationExecutor.executeOp()` is supposed to handle Ether transactions, hence it has been changed to `payable`.

6.9 Reentrancy Into `executeOp()`

Design | Medium | Version 1 | Code Corrected

Function `executeOp()` can be reentered. At this time OperationStorage may be in an inconsistent state, amongst others (`actions`), `returnValues` may contain values.

While this is not an intended use case, technically the possibility exists. To reduce risks, this may be restricted especially as the execution reaches untrusted third-party code (integrations, token contracts).

Furthermore note that after a `takeAFlashloan` action, the OperationExecutor temporarily has the right to call `execute()` on the DsProxy of the user. `OperationExecutor.onFlashloan()` uses this to execute `aggregate()` on the `initiator`.

Currently this is not exploitable due to:

- The DAI Flash Mint Module features a reentrancy protection, hence no second flashloan is currently possible. Note that this is no requirement for an ERC3165 compliant flashloan provider, an arbitrary flashloan proivder may not, e.g., the reference implementation of ERC3165 does not feature such a protection.
- ERC3165 requires the `initiator` being the `msg.sender` initiating the flashloan. It's not possible for an attacker to get the initiator to be the DsProxy where the OperationExecutor holds the privilege.

In the Version 1 reentrancy is also possible with `aggregate()`, please consider issue Visibility of aggregate function.

Code corrected:

The updated code prevents reentrancy into `executeOp()` by leveraging the OperationStorage contract: The reentrancy lock is set in the OperationStorage at the beginning of the execution and released after the operation.

Releasing can only be done by the account which set the reentrancy lock; releasing the lock sets the stored account to `0x0`. This ensures that the original call to executeOp() reverts in case of reentrancy.

6.10 Visibility of `aggregate` Function

Design Medium Version 1 Code Corrected

Although the main entry point into the `OperationExecutor` contract is the `executeOp` function, the `aggregate` function is also `public`. In the current implementation this is required for the flashloan functionality to continue execution of the subsequent calls.

This function, which is not intended to be called directly, may become a source of confusion/errors. In particular, if called directly it will bypass the verification that the right actions are executed for a given operation (as specified by `OperationsRegistry`). Furthermore `operationStorage.finalize()` will not be executed.

Access to this function might be restricted. This function may be `internal`, with an exposed `external` function for `onFlashloan()` which accepts calls by the OperationExecutor only.

Code corrected:

The visibility of the `aggregate` function has been changed to `internal`. The callback from `onFlashLoan` to the DsProxy is executed via a new `callbackAggregate` function which is public but restricts execution only by OperationExecutor itself.

6.11 sendToken: Transfers `msg.value` Instead of `send.amount`

Correctness Medium Version 1 Code Corrected

In case of Ether transfer, action sendToken transfers `msg.value` instead of the amount specified in `sendTokenData`.

Note that despite the `payable` modifier of function `execute`, (delegate)calls from the OperationExecutor to this action cannot have a non-zero `msg.value` since `OperationExecutor.aggregate()` is not

payable and would revert on non-zero `msg.value`. Neither does `executeOp()`. Hence `sendToken` will never enter the `msg.value > 0` branch in the current setup.

Code corrected:

In case of Ether transfer, action sendToken now transfers the amount specified in `sendTokenData`. Furthermore `OperationExecutor.executeOp()` now features the `payable` modifier and accepts calls with Ether. Since function `aggregate` has been made `internal` calls with non-zero `msg.value` are now supported.

6.12 Action Events Not Emitted

Design Low Version 1 Code Corrected

The `Executable` interface defines an `Action` event that only some actions emit. The following actions do not emit an event at the end of their execution:

- `common/PullToken`
- `common/SendToken`
- `common/SetApproval`
- `common/SwapOnOneInch`
- `maker/CdpAllow`

Code corrected:

All actions now emit the `Action` event.

6.13 Actions: Inconsistent Destination of Tokens

Design Low Version 1 Code Corrected

In maker/Generate the destination address `data.to` can be specified by the caller, but this is not possible in aave/withdraw. There may be a general pattern actions should adhere to for consistency.

Code corrected:

AAVE withdrawal & borrow actions now accept the destination of tokens, consistent with the Maker actions.

6.14 DAI Address Could Be Constant

Design Low Version 1 Code Corrected

In TakeFlashoan, the DAI address may be hardcoded instead of being queried from the registry.

Code corrected:

The DAI address is now an immutable set upon deployment.

6.15 OperationStorage: Unused `owner` Variable

Design Low Version 1 Code Corrected

The OperationStorage contract defines an `owner` variable that is set to `msg.sender` in the constructor, but is never used thereafter.

Code corrected:

The unused owner variable has been removed.

6.16 OperationsRegistry: No Events Emitted on State Change

Design Low Version 1 Code Corrected

No events are defined or emitted in the OperationsRegistry contract. In general, it is recommended to emit an event on every state change. This allows to identify changes easily.

Code corrected:

Function `addOperation` now emits event `OperationAdded`.

6.17 Outdated Compiler Version

Design Low Version 1 Code Corrected

The project uses an outdated version of the Solidity compiler.

```
pragma solidity ^0.8.5;
```

Known bugs in version 0.8.5 are:

https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1753

More information about these bugs can be found here: https://docs.soliditylang.org/en/latest/bugs.html

At the time of writing the most recent Solidity release is version `0.8.16`.

Code corrected:

It was decided to update to compiler version `0.8.15`. Client states:

```
Updated compiler version to 0.8.15. There was a new version
after this that came just after we updated, but that did not have changes that were
relevant to our scope.
```

6.18 Receiver of Flashloan

Design | Low | Version 1 | Code Corrected

From the documentation we understood that the receiver of the flashloan and the callback would always be the OperationExecutor contract. For action `TakeFlashloan` however, the caller can specify the receiver using parameter `flData.borrower`. What's the intention here?

Code corrected:

The address of the OperationExecutor is now fetched from the registry and set as borrower.

6.19 Sanity Check in on Flashloan

Design | Low | Version 1 | Code Corrected

The intention behind following check in `OperationExecutor.onFlashloan()` is unclear:

```
require(amount == flData.amount, "loan-inconsistency");
```

While checking the actual balance has its limitations (e.g. not clear if token balance originates from the flashloan or whether it has been there before already), why is the sanity check on the specified amounts being done but not on the actual balance ?

After the intermediate report, the check was changed to:

```
require(IERC20(asset).balanceOf(address(this)) == flData.amount, "Flashloan inconsistency");
```

This is dangerous: Any additional balance of this token held by the OperationExecutor causes a revert of this function. It should be clarified what should be checked and why this is checked.

Initially the code checked whether the parameter `amount` the caller (the flashloan provider) passed matches the expected amount. We questioned what's the intention behind this check and highlighted that it doesn't ascertain anything on the actual balance. Checking if at least the balance expected is present might be an option, but it must be clear that this doesn't say how much tokens have been transferred from the flashloan provider (as the OperationExecutor may have had a non-zero token balance before as anyone could just transfer tokens).

Code corrected:

The check was changed to:

```
require(IERC20(asset).balanceOf(address(this)) >= flData.amount, "Flashloan inconsistency");
```

This does not allow to determine whether the token balance originates from the flashloan or if it was already in the contract, but now an additional balance of this token held by the OperationExecutor will not cause a revert.

Client adds:

```
In the event that the lender is compromised and supplies a lesser amount then the
Operation execution will fail unless another party has accidentally sent balance to
```

```
the Operation Executor at some earlier time. The only impacted party would be the person
who accidentally sent tokens to the Operation Executor, whose funds are lost anyway.
```

6.20 Swap Slippage Saved Event Order

Design Low Version 1 Code Corrected

`Swap._swap()` emits the `SlippageSaved` event after a swap:

```
balance = IERC20(toAsset).balanceOf(address(this));
emit SlippageSaved(receiveAtLeast, balance);
if (balance < receiveAtLeast) {
  revert ReceivedLess(receiveAtLeast, balance);
}
```

While after a revert occurs all state changes including any event logs are thrown away, it might be more appropriate to only emit the event after it has been ascertained that more than `receiveAtLeast` have been received.

Code corrected:

The event is now emitted after the check.

6.21 Swap.sol: ReceiveAtLeast Does Not Take Into Account the Fee

Correctness Low Version 1 Specification Changed

While the code of `Swap._swap()` ensures that after the call to the 1inchAggregateor the balance of the SwapContract is more than `receiveAtLeast` what is sent onwards to the user might be less as the fee may be deducted only afterwards. The expected behavior is not specified.

Specification changed:

The documentation has been updated and now reads:

```
receiveAtLeast - an amount that needs to be returned from swap, it does not consider fee,
in case fee is collected from outgoing token the resulting amount might be less than receiveAtLeast.
Sole purpose of ReceiveAtLeast is to prevent high slippage on exchange.
```

6.22 Unused Return Value of Aave Withdraw

Design Low Version 1 Code Corrected

In the Aave withdraw action, the return value of `withdraw` is ignored:

```
ILendingPool(registry.getRegisteredService(AAVE_LENDING_POOL)).withdraw(
  withdraw.asset,
  withdraw.amount,
  address(this)
);
```

This return value represents the actual amount that was withdrawn and might be different from the amount given as argument.

Specifically, if `type(uint256).max` is given as argument `amount`, then the total available balance is withdrawn and returned.

Code corrected:

The withdrawn value is now stored and can be used by subsequent actions.

Oazo Apps Limited replied:

```
We agreed that all Actions should push a
return value to the OperationStorage even if that value is zero (Code change hasn't
occurred yet). This makes paramsMapping simpler and more predictable.
```

6.23 `onFlashLoan()` Ignoring Fees

Correctness **Low** **Version 1** **Code Corrected**

The documentation states that the OperationExecutor implements the IERC3156 standard. The implementation of `OperationsExecutor.onFlashloan()` however does not support fees. DAI flash mint currently takes no fee hence with the intended flashloan provider the code currently works however the current code doesn't fully implement/support the ERC3156 standard.

Code corrected:

`onFlashloan()` now supports fees.

7 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.

7.1 DsProxy With Unsupported Authority

Note Version 1

A user is free to set the `authority` contract of his own DSProxy. Depending on the authority contract set, which may be arbitrary, `ProxyPermission.givePermission()` may not be successful. The documentation only explains the expected case were everything works, it does not mention this restriction.