PUBLIC

Code Assessment

of the DeFi Modular Actions v2 Smart Contracts

August 07, 2023

Produced for

summer.fi

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Summer.fi with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of DeFi Modular Actions v2 according to Scope to support you in forming an opinion on their security risks.

Summer.fi implements an updated and more gas-efficient version of modular proxy actions. The system allows multiple actions to be executed from a UserProxy in a single call.

The most critical subjects covered in our audit are functional correctness, access control and integrations with external systems. Functional correctness is high. One issue concerning Access control has been resolved after the intermediate report. Security regarding integration with external systems is high.

The general subjects covered are gas efficiency, documentation and testing. Gas efficiency is good and is a significant improvement over the previous version. The documentation provided is satisfactory. The available tests covering v2 are very basic only, we strongly recommend to improve the test coverage.

In summary, we find that the codebase provides a good 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	1
• Code Corrected	1
Medium-Severity Findings	0
Low-Severity Findings	3
• Code Corrected	3

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 DeFi Modular Actions v2 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	19 June 2023	29e6cbaeb3ef9c7147d400a38d7a8c3a66b35946	Initial Version
2	24 July 2023	3ca95f9f80ff123859ea495714733b06a5533a05	After Intermediate Report
3	07 August 2023	f75a7bbd6977497d3ef10ab6b257383298058a58	Updated SendToken

For the solidity smart contracts, the compiler version `0.8.15` was chosen.

The following files were in scope:

```
dma-contracts/contracts/v2/OperationExecutor.sol
dma-contracts/contracts/v2/OperationRegistry.sol
dma-contracts/contracts/v2/UseStorageSlot.sol
dma-contracts/contracts/v2/actions/*
```

2.1.1 Excluded from scope

All other files are out of scope.

External Systems such as Aave, Balancer and Maker are expected to work correctly as documented.

2.2 System Overview

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

DeFi Modular Actions V2 is an updated and more gas-efficient version of the previous modular proxy actions. The system allows executing a series of calls aggregating actions (called an operation). 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. Action addresses are queried from the trusted ServiceRegistry. This concept allows updating or adding new actions seamlessly. Only sequences of actions that have been whitelisted by being added to the OperationsRegistry can be executed.

Currently, the following actions are implemented:

- **SendTokenV2:** Sends an amount of tokens or transfers `msg.value` amount of Ether to the given address.
- **SetApprovalV2:** Gives approval to transfer an amount of an asset to the given address.
- **SwapActionV2:** Gives approval and calls `swapTokens()` on the swap contract (fully trusted, expected to return the swapped tokens correctly). Writes the amount received into `returnedValues`.
- **TakeFlashloanV2:** Takes a flashloan. This version supports Balancer (various tokens supported but only one can be borrowed per flashloan action) and Maker's Flash Mint Module (DAI only). Gives permission to the OperationExecutor to call `Execute(address,bytes)` on the current proxy account executing. The proxy used must be compatible. Technically, permission is given using `DsGuard.permit()`: A DPMProxy has a guard by default. For DsProxy, a new DsGuard is deployed if no authority is set. The execute permission is revoked afterwards. The OperationExecutor is called back from the flashloan provider, which calls into the proxy context again to execute the calls nested in the flashloan. Tokens to repay the flashloan must be made available, the code of the flashloan action then orchestrates the repayment.
- **UnwrapEthV2:** Unwraps WETH into Ether. If `type(uint256.max)` is specified as amount, the account's balance is used as amount.
- **WrapEth:** Wraps Ether into WETH. If `type(uint256.max)` is specified as amount, the account's balance is used as amount.

For Aave V3:

Note that some of the action names differ from the names used within Aave.

- **AaveV3BorrowV2:** Borrows from Aave v3 by executing `AAVE_POOL.borrow(borrow.asset, borrow.amount, 2, 0, address(this))`: Borrows `borrow.amount` of `borrow.asset` on behalf of `address(this)` (the proxy executing). The interest rate mode parameter is hardcoded to 2 (variable). The referral code is set to 0. Stores the amount borrowed into `returnedValues`.
- **AaveV3DepositV2:** Deposits into Aave v3 and stores the deposited amount into `returnedValues`. The amount to be deposited can either be specified or can be read from the `returnedValues` of previous calls. Executes `AAVE_POOL.supply(deposit.asset, actualDepositAmount, address(this), 0)` to supply `actualDepositAmount` of `asset` on behalf of `address(this)` (the proxy executing). The referral code is set to 0. **Note:** the pool must have sufficient allowance to spend the funds of `msg.sender`. Finally, depending on the flag `setAsCollateral`, may set the asset to be used as collateral.
- **AaveV3PaybackV2:** Repays an asset into the Aave V3 pool set for this action. The amount can either be specified (where `type(uint256).max` is used as magic value to use the available balance) or can be read from the `returnedValues` of a previous call. Stores the amount repaid into `returnedValues`. **Note:** the pool must have sufficient allowance to spend the funds of `msg.sender`.
- **AaveV3SetEModeV2:** sets the user's eMode (user efficiency mode category) on the Aave Lending pool set for this action. Stores the EmodeData into `returnedValues`.
- **AaveV3WithdrawV2:** Withdraws the specified amount by redeeming aTokens. Stores the amount withdrawn into `returnedValues`.

Actions are combined as desired in so-called operations.

The calls of the operation to be executed consists of an array containing the individual action calls. An action call consists of the target hash and the calldata. The calldata is made of the action data (depends on the action, please refer to *core/types/common.sol* for the individual action data) and the paramsMap

`uint8[] memory paramsMap`. ParamsMap can be used by the code of the action to replace parameters of the action data with the return value of a previously executed action.

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.

OperationExecutorV2

This is the main contract. It features the following entry points:

- `executeOp()`: The user will execute this function through his DsProxy / DPM Proxy as delegatecall. It executes the calls and records them. In the end, the action order executed is checked against a set of legal operations. If the actions do not form a legal operation, they will revert. Unlike OperationExecutor V1, all actions are now considered. In V1, nested actions (e.g. within a Flashloan) were not considered part of an operation. Additionally, the check is now done after the actions execute, whereas in V1 the operation was defined at the beginning of execution.
- `onFlashloan()`: Implements the interface for the callback of the ERC3156 compliant flash loan provider. Processes the remaining calls via the Initiator (Proxy).
- `receiveFlashLoan()`: Implements the interface for the callback of the Balancer flash loan. Processes the remaining calls via the Initiator (Proxy).
- `callbackAggregate()`: Is called by the OperationExecutor via the proxy to execute actions nested in a FlashloanAction.

Return values of actions are stored within the context of the proxy starting at storage slot `(keccak256("proxy.transaction.storage")) - 1)`.

The new AutomationBot may use the OperationExecutor. Through appropriate commands it executes `executeOp()` through a proxy; from the OperationExecutor perspective the flow of execution does not differ compared to user execution through proxies. Contrary to previous versions flashloan callbacks are no longer executed in the context of the OperationExecutor but switch back into the context of the AutomationBot command's proxy.

OperationsRegistryV2

This contract keeps a mapping of operations identified by a string and is used to verify that a sequence of actions executed is a valid operation. The mapping can be updated by the owner. Should the name exceed 64 characters, the name is simply truncated.

`getOperationName()`: This function returns either the name of the operationHash or reverts. Used by the OperationExecutor to check whether the sequence of actions was valid. Note that only the order of actions is checked, not their arguments (action calldata).

This contract is not to be confused with the ServiceRegistry (called Registry within the OperationExecutor) of Summer.fi, which is used as source of truth to retrieve the implementation addresses of actions.

The owner can update the owner.

2.3 Trust Model & Roles

Frontend: Fully trusted to craft a correct set of calls, including proper calldata.

Summer.fi: Fully trusted to operate the OperationsRegistry correctly and to add genuine and correct actions executing as documented only. The ServiceRegistry managed by Summer.fi is fully trusted.

User: Untrusted, any user may interact with the contracts through the context of their own DSProxy or DPM Proxy. 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. This also includes trusting the third-party APIs used by the Summer.fi front-end.

Users may also interact with the operation executor directly. **This is not an intended use case and should not be done.** As the contract is stateless this should not have an impact.

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

This version of the OperationExecutor will store temporary data in the executing account's storage starting at slot `keccak256("proxy.transaction.storage")-1`. There is an assumption that this does not collide with any other stored data.

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	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	1
Medium-Severity Findings	0
Low-Severity Findings	3

- Balancer Flashloan Can Break Proxy Access Rights Code Corrected
- Inefficient txStorage Code Corrected
- SafeMath Is Not Needed Code Corrected
- SendToken Functionality Differs for Native Tokens Code Corrected

6.1 Balancer Flashloan Can Break Proxy Access Rights

Security High Version 1 Code Corrected

CS-DMAV2-005

The OperationExecutor supports Flashloans. For this, a callback interface for the FlashloanProvider must be provided. As the UserProxy does not have the required interface, this callback is made to the OperationExecutor, which then calls `execute()` to switch back into the Proxy's account/storage context

The transaction flow for Flashloans is as shown below:

Context: UserProxy

UserProxy delegatecall -> OperationExecutor -> TakeFlashloan Action -> FlashloanProvider

Context: OperationExecutor

FlashLoan Callback -> UserProxy.execute()

Context: UserProxy

OperationExecutor.callbackAggregate()

To facilitate this flow, the OperationExecutor is given the permission to call `execute()` on the UserProxy for the duration of the Flashloan.

In the case of using Maker's Flashloan, this callback is correctly restricted to only call `execute()` on the `initiator` (msg.sender) of the Flashloan.

For the Balancer integration however, the `initiator` is given in calldata of the Balancer call instead of Balancer returning the `msg.sender`. This means there can be a callback into any proxy for which the OperationExecutor has execution rights, even if Balancer was called from an address other than the Proxy.

This is an issue if there is an unsafe external call in any action nested in the FlashLoan. Any address can reenter the `receiveFlashloan` function of OperationExecutor by taking a Flashloan on Balancer, and use its `execute()` privileges to execute any actions from within the context of the proxy.

We illustrate what an attack could look like using the following set of actions performed by a user:

1. TakeFlashloan
2. SwapAction
3. TransferAction

First, the user will take a Flashloan. This will give the OperationExecutor permission to call `execute()` on the proxy. Second, the user swaps tokens. Assume that this makes an unsafe external call to an attacker's contract, for example through a transfer hook in the traded token.

Now, the attacker has control flow and can execute the reentrancy attack. They take another Flashloan, setting the `FlashloanRecipient` to the OperationExecutor and the `initiator` to the UserProxy. `receiveFlashloan()` will be called, which calls `execute()` in the UserProxy. The functionSelector will always be `callbackAggregate`, but the `flData.calls` is provided by the attacker when they take their Flashloan. Now, those actions chosen by the attacker will be executed in the context of the UserProxy. For example, the attacker can call the SendToken, SetApproval or Withdraw action and drain all funds in the proxy.

The actions taken by the attacker will be recorded in the txStorage and it will be checked if they belong to an operation that exists at the end of the execution. This means the attack only works if there exists an action that is the same as the one the user called, but with additional actions added.

For example, for the set of actions above to be attackable there would have to be another legal operation that has those steps and additional ones at the location of the reentrancy.

Such as:

1. TakeFlashloan
2. SetApproval
3. SwapAction
4. TransferAction

The same issue is also present in OperationExecutor v1, when taking a Balancer Flashloan and making an unsafe external call. However, here the limitation of the actions needing to form a legal operation is not present, as the operations are only checked in the beginning, not at the end. So an attacker can add any operations they want using reentrancy.

In summary, if there is an unsafe external call (reentrancy) in any nested action taken by a user during a Balancer Flashloan, their entire Proxy's balance can be drained.

Code corrected:

Nested flashloans are now prevented, this solves the main concern of reentering the OperationExecutorV2 at a time when it has execution rights on the Proxy of the original caller.

Technically this works as follows: OperationExecutorV2 now features a flag `isFlashloanInProgress`. Upon a callback through one of the flashloan interfaces (the callback by the flashloan provider is executed in the context of the OperationExecutorV2) `checkIfFlashloanIsInProgress` ensures no flashloan is in progress already by ensuring the flag is equal to 1. `processFlashloan()` sets the flag to 2 before dispatching the callback to the Proxy context and resets the flag to 1 after it returned.

Note that outisde of flashloan actions, the following still applies:

```
For the Balancer integration however, the ``initiator`` is given in calldata
of the Balancer call instead of Balancer returning the ``msg.sender``.
```

It is important that outside of flashloan actions the OperationExecutorV2 does not have any priviledges on any proxy.

6.2 Inefficient txStorage

Design Low Version 1 Code Corrected

CS-DMAV2-006

In OperationExecutor's `aggregate()`, actions are written into txStorage as follows:

```
txStorage.actions = abi.encodePacked(txStorage.actions, targetHash);
```

This unnecessarily reads all past actions out of storage and then rewrites the same data into storage again, which costs gas each time. To reduce gas consumption, just the new data could be added at the end.

Writing to (multiple) storage slots is expensive. It would also be possible to hash actions one by one instead of at the end, allowing only a single storage slot to be used instead of one for each action.

Code corrected:

`aggregate()` now pushes the executed action to the storage without reading the previous actions every time:

```
txStorage.actions.push(targetHash);
```

`executeOp()` retrieves the stored actions once after the execution of `aggregate()` completed and encodes them, the result is the same as the previous `txStorage.actions`.

6.3 SafeMath Is Not Needed

Design Low Version 1 Code Corrected

CS-DMAV2-003

An outdated version of the SafeMath library is used.

Since Solidity 0.8, the overflow checks that were previously done in SafeMath are now enforced by default. As a result, the library is no longer needed.

If SafeMath is used for the custom revert strings, the new version of SafeMath should be used.

Code corrected:

SafeMath has been removed.

6.4 SendToken Functionality Differs for Native Tokens

Correctness Low Version 1 Code Corrected

CS-DMAV2-004

The SendToken action has the following NatSpec:

```
//@title SendToken Action contract
//@notice Transfer token from the calling contract to the destination address
```

For ERC-20 tokens, the implementation sends tokens that are held by the delegatecalling contract.

However, for native ETH, the implementation only forwards ETH that was sent as `msg.value` along with the originating function call. In this case `send.amount` is ignored. It cannot send ETH that was already held by the delegatecalling contract.

After the intermediate report the description was updated and now explains the different behavior for ERC-20 tokens and Ether.

The description for Ether is inaccurate however:

```
The amount of ETH that can be transferred is either the whole or partial
(whether some amount has been used in other actions) amount from the
amount that the transaction has been called with ( msg.value ). If the proxy
contract contains any prior ETH balance, it CANNOT be transferred.
```

The delegatecalls into the code of the actions during the loop in `aggregate()` preserve `msg.value`. `payable(address).transfer(msg.value)` attempts to transfer this amount which succeeds only if sufficient Ether is available. Either the Ether sent along the original call has not yet been spent and is available for the onward transfer or the Proxy has additional Ether balance which can be used.

Code corrected:

SendToken has been changed, the behavior for the native token now matches the functionality for ERC-20 tokens. The new description now describes the actual behavior of the action.

6.5 ERC-1967

Informational | Version 1 | Specification Changed

CS-DMAV2-002

The description of library StorageSlot reads:

```
This library is a small implementation of EIP-1967. Unlike that EIP which usage
is to store an address to an implementation under specific slot, it is used to
storage all kind of information that is going to be used during a transaction
life time.
```

While the library is suitable for the intended use, mentioning EIP-1967 can be confusing and strictly speaking is incorrect: This EIP standardises the storage slots for the the following addresses: implementation, beacon and admin only. The EIP states:

```
More slots for additional information can be added in subsequent ERCs as needed.
```

The StorageSlot library borrows the idea how the storage slot is chosen to avoid any collision with high probability from the EIP-1967 but is otherwise not connected to and does not implement or adhere to EIP-1967.

Specification changed:

Summer.fi removed the misleading reference to ERC-1967 from the description.

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 Missing Event

Informational **Version 1** **Acknowledged**

CS-DMAV2-001

In OperationsRegistry, the `transferOwnership` function does not emit an event.

Acknowledged:

Summer.fi states they do not emit events on ownership change in any contract.

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 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.

8.2 Reentrancy Could Delete Transient Storage

Note Version 1

The `executeOp` function in OperationExecutor could be reentered, which deletes the txStorage.

However, this is only possible if the `msg.sender` has `execute()` privileges for the `executeOp` function on the UserProxy and can reenter.

This should only be the case for the user, meaning they could only circumvent the legal operations check for themselves by deleting txStorage.