PUBLIC

Code Assessment

of the DMA v2 II Smart Contracts

June 26, 2024

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 DMA v2 II according to Scope to support you in forming an opinion on their security risks.

Summer.fi implements updates to the DeFi Modular (DMA) Actions v2 architecture to support the usage of transient storage. See the previous report for reference.

The most critical subjects covered in our audit are the usage of transient storage and functional correctness. Functional correctness is improvable due to incorrect data being written, see Aave V3 actions bad data written. Additionally, there could be reentrancy scenarios in bad setups, see Reentrancy Into the Contract. In case governance is untrusted, governance could add contracts such that this could be exploited. Further, the design is improvable due to Collisions on Operations.

The general subjects covered are documentation, trustworthiness and gas efficiency. Documentation is improvable, see Unclear actions setup. Trustworthiness is satisfactory. However, it is improvable, see the paragraph above.

In summary, we find that the codebase provides a satisfactory but improvable level of security.

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	10
• Code Corrected	4
• Risk Accepted	5
• Acknowledged	1

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the source code files inside the DMA v2 II 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	15 April 2024	f3c903c534fb14dd9c8118c8bed3155495d8c9c2	Initial Version
2	03 June 2024	56558d6d02f14ed0c5d4dc28f785fd40a39cd0d6	After Intermediate Report
3	11 June 2024	770123377bb08cc5dc252e8eb054a04d90da5092	Further fixes
4	25 June 2024	ce0d7b8b773a8c0a526937f2a31b67928d40adba	Additional fixes

For the solidity smart contracts, the compiler version `0.8.24` was chosen.

The files below were in scope core system:

```
contracts/libs/Address.sol
contracts/libs/SafeMath.sol
contracts/libs/SafeERC20.sol
contracts/libs/ActionAddress.sol
contracts/libs/UseStorageSlot.sol
contracts/libs/UseRegistry.sol
contracts/libs/DS/ProxyPermission.sol
contracts/core/OperationExecutor.sol
contracts/core/OperationsRegistry.sol
contracts/core/types/Common.sol
contracts/core/constants/Aave.sol
contracts/core/constants/Common.sol
contracts/core/constants/Balancer.sol
contracts/core/types/Aave.sol
contracts/interfaces/aaveV3/DataTypes.sol
contracts/interfaces/aaveV3/IPoolAddressesProvider.sol
contracts/interfaces/aaveV3/IPoolV3.sol
contracts/actions/common/Executable.sol
contracts/interfaces/flashloan/IERC3156FlashBorrower.sol
contracts/interfaces/flashloan/balancer/IFlashLoanRecipient.sol
contracts/interfaces/tokens/IERC20.sol
contracts/core/constants/Maker.sol
contracts/interfaces/tokens/IWETH.sol
contracts/actions/common/WrapEth.sol
contracts/actions/common/UnwrapEth.sol
contracts/interfaces/balancer/IVault.sol
contracts/actions/common/TakeFlashloan.sol
contracts/interfaces/flashloan/IERC3156FlashLender.sol
```

```
contracts/actions/common/SwapAction.sol
contracts/actions/common/SetApproval.sol
contracts/actions/common/SendToken.sol
```

2.1.1 Excluded from scope

All other files are out of scope. Any potential action not in scope or any integration with the contracts is out of scope. All external systems are out of scope. Some files define constants that are undocumented; the selection of these constants is out of scope.

2.2 System Overview

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

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Summer.fi implements updates to the DeFi Modular (DMA) Actions v2 architecture. Note that a previous version has been audited in an earlier report. The main change introduced compared to the initial version is that transient storage is now used for storing data necessary during a transaction.

2.2.1 Core

The DMA system allows executing a set of actions as a batch; called operations. As a central entry point, the `OperationExecutor` offers the `executeOp(Call[] memory calls)` function. The function is intended to be called with a `delegatecall` and thus is executed in the execution context of the proxy executing the call. It fetches the action contract addresses from the Summer.fi's service registry and performs delegatecalls to the action contracts `execute(bytes calldata data, uint8[] memory paramsMap)` function.

Ultimately, the `OperationExecutor` acts as a multicall implementation leveraging the service registry as a source for logic templates to integrate with DeFi protocols. Note that actions may use data from previous subcalls of the multicall. While that was possible in the previous version of DMA v2, the mechanism to achieve this has been adapted to leverage transient storage.

Further, transient storage is used to keep track of the chain of calls so that the set of executed actions (operation) can be validated against the `OperationsRegistry`. Namely, an operation is defined as the `keccak256` of the packed encoding of the service name hashes of the actions.

Note that arrays in transient storage are used. Namely, they are encoded as follows:

1. The storage slot `slotPosition` of an array with the `string` identifier `salt` is: `keccak256(abi.encodePacked("summer.proxy.storage", salt))`
2. The length of an array with `slotPosition` computed in 1. is stored at `keccak256(abi.encodePacked(slotPosition, "length"))`
3. The item at position `i` is stored at slot `keccak256(abi.encodePacked(slotPosition, i))`.

The transient arrays `"actions"` and `"transaction"` are used for storing the set of executed actions and the data of previous calls, respectively.

The `OperationExecutor` implements the EIP-3156 callback function `onFlashLoan` and the Balancer callback function `receiveFlashloan` to be able to receive flashloans. Note that a flashloan proceeds to move the flashloaned funds to the proxy and aggregates further actions by calling the proxy so that it can `delegatecall` into `callbackAggregate(Call[] memory calls)`. That starts a new multicall

sequence similar to `executeOp`. However, the operation is not verified as the executed sub-actions of the flashloan action are already validated as part of the action chain of the top-level `executeOp` operation.

2.2.2 Actions

Actions implement an interaction with a protocol (e.g. borrowing on Aave). They implement a `parseInputs(bytes memory _callData)` function that decodes the input arguments for the integration. Further, `execute(bytes calldata data, uint8[] memory paramsMap)` is implemented for each action to allow the `OperationExecutor` to forward the `delegatecall`.

All actions should follow the following pattern:

1. `parseInputs`: Parse the input data for the integration.
2. Adapt the input as specified: Based on `paramsMap`, adapt the input data for the integration. Namely, that reads from transient storage to retrieve values previously used (e.g. the amount WETH minted should be the amount of WETH to deposit into Aave). Note values (transient storage slots) in `paramsMap` equal to `0` should use the input data from `parseInputs`. Thus, when they are some non-zero `i`, the array item at `i-1` of the `"transaction"` array in transient storage should be used instead of the passed parameter.
3. Execute the action: Given the adapted input, perform the call to the external system.
4. Push the data to the `"transaction"` array in transient storage so that it can be referenced by subsequent actions.

The following general actions are implemented

1. `WrapEth` wraps ETH. `paramsMap` at 0 can be used to read the amount to wrap from transient storage. If the amount is `uint256.max`, the full balance is wrapped.
2. `UnwrapEth` unwraps wETH. `paramsMap` at 0 can be used to read the amount to unwrap from transient storage. If the amount is `uint256.max`, the full balance is unwrapped.
3. `SetApproval` gives a token allowance to an address (either setting allowance or increasing allowance). `paramsMap` at 2 can be used to read the amount for the operation from transient storage.
4. `SendToken` and `SendTokenAuto` can send a given amount of a token or ETH to a recipient. `paramsMap` at 2 can be used to read the amount to send from transient storage (`SendTokenAuto` always reads from transient storage). If the amount is `uint256.max`, the full balance is sent.
5. `SwapAction` swaps tokens on the 1inch wrapper of Summer.fi. The received amount is pushed on the transient storage array.

The following special actions (granting approval) are implemented:

1. `TakeFlashloan` takes a flashloan from Balancer V2 or the Maker flashmint module. **Note that this temporarily grants permission to the executor due to the callback being directed there so that the flashloan hooks can continue the aggregation (as described above).

The following are the Aave v3 actions:

1. `AaveV3Borrow` borrows a given amount of asset from Aave v3. `paramsMap` at 1 can be used to read the borrowed amount from transient storage. The borrowed amount is pushed on the transient storage array.
2. `AaveV3Deposit` supplies an amount of an asset to Aave v3. `paramsMap` at 1 can be used to read the amount to deposit from transient storage (note that this amount is either added to the amount defined as a parameter or fully used to replace the parameter). The supplied amount is pushed on the transient storage array.

3. `AaveV3Payback` repays an amount of token debt on Aave v3. `paramsMap` at 1 can be used to read the amount to repay from transient storage. The repaid amount is pushed on the transient storage array.
4. `AaveV3SetEMode` sets the e-mode category. The set e-mode category is pushed on the transient storage array.
5. `AaveV3Withdraw` withdraws an amount of tokens to an address from Aave v3. The withdrawn amount is pushed on the transient storage array.

2.2.3 Roles and Trust Model

The following roles are defined:

1. Front-End: Trusted to provide proper data. The front-end is of utmost importance. It is required to provide correct, reasonable and valid data. However, users bear the responsibility to validate the transactions.
2. Summer.fi: Untrusted. While governance can whitelist operations and actions, it should not be able to steal user funds.
3. Users: Untrusted. Users are untrusted. However, the users are expected to understand the operations they are executing. Namely, the user should be aware of the potential drawbacks of actions.

The general expectation is that the system is interacted with through user-owned proxies through delegatecalls. Additionally, it is expected that the executor never holds any permissions except for flashloan actions (similarly no action should hold any privileges). Interacting with actions that hand out any sort of privilege can be dangerous and should be considered carefully.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 Findings

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

- Design: Architectural shortcomings and design inefficiencies
- 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	6

- Aave V3 Actions Bad Data Written Risk Accepted
- Collisions on Operations Risk Accepted
- Hypothetical Collision on Storage Risk Accepted
- Reentrancy Into the Contract Risk Accepted
- Unclear Actions Setup Acknowledged
- Unlimited Repayment on Behalf of Others Risk Accepted

5.1 Aave V3 Actions Bad Data Written

Correctness Low Version 1 Risk Accepted

CS-DMAv2II-001

The Aave V3 actions may write wrong data to transient storage. Namely,

1. Deposit: Aave has an off-by-one error that may occur (due to rounding) when supplying collateral. The received amount of aTokens could differ from the amount of tokens deposited.
2. Payback: If `paybackAll`, the unused `amount` is written to transient storage. If it is false but `amount` is `uint.max`, the amount actually paid back is not written but `uint.max` is.

Risk accepted:

Summer.fi accepts the risk and states:

```
We will keep that in mind on the next deployment of action contracts.
```

5.2 Collisions on Operations

Design Low Version 1 Risk Accepted

CS-DMAv2II-003

An operation must be registered in the registry by its hash which is the hash of the packed encoding of the actions in the order they are executed. However, due to a lack of depth of calls, two similar but distinct operations may have the same hash. However, note that the same operations must be used. Consider the below example traces that are semantically distinct but would share the same operation hash.

1. Execution 1:
    1. Action 1: Flashloan action
        1. Action 2: During the flashloan action 2 is executed.
2. Operation 1 is executed:
    1. Action 1: Flashloan
    2. Action 2: After the flashloan, action 2 is executed.

The encoding for both would be `Action 1 | Action 2`, there is a fundamental difference between the two.

Risk accepted:

Summer.fi accepts the risk and states:

```
We accept this risk in the current implementation due to the limitations in our encoding strategy.
However, we are committed to enhancing the robustness of our operation executor. In future versions,
we plan to redesign the operation executor to incorporate a more granular encoding method that
captures the depth and context of each action within an operation, thereby preventing hash
collisions for semantically distinct operations.
```

5.3 Hypothetical Collision on Storage

Correctness **Low** **Version 1** **Risk Accepted**

CS-DMAv2II-004

The length of an array is stored at the position

```
keccak256(abi.encodePacked(slotPosition, "length"))
```

while items are stored at the position

```
keccak256(abi.encodePacked(slotPosition, index));
```

If `bytes32("length") == bytes32(index)`, a storage collision may occur for an array (however, unlikely due to a high index being needed).

Further, note that it does not follow the Solidity storage encoding, see Solidity docs.

Risk accepted:

Summer.fi accepts the risk and states:

```
While we accept the auditor's finding, we assess the practical risk of such collisions to be
minimal.
```

5.4 Reentrancy Into the Contract

Design | Low | Version 1 | Risk Accepted

CS-DMAv2II-005

The `OperationExecutor` should typically never hold permissions unless a flash loan is executed (proxy permissions are given). Thus, during flashloans the executor offers an attack surface for reentrancy. While the flashloan functions are protected from being reentered, others are not. Namely, `executeOp` and `callbackAggregate` are unprotected. Further, they can be invoked directly with a call.

Consequently, calling the unprotected functions directly may lead to reentrancy issues. However, that would require a special setup with actions. Consider the following example:

1. A user takes a flashloan
    1. and proceeds to swap some tokens.
        1. Then, the exchange contract does a callback to the attacker (e.g. native ETH transfer)
            1. where the attacker invokes `executeOp` with a hypothetical action that allows calling `execute` on a proxy.
        2. The attacker had access to a proxy.
    2. Then, the user proceeds to perform some further actions.

Ultimately, given suitable actions, the ability to call proxy function with `call` may lead to issues. Additionally, the permissions during flashloans are held during the full lifespan of the proxy. However, the proxy could try to already revoke the permissions earlier.

Risk accepted:

Summer.fi accepts the risk and states:

```
We will further reassess the possibility of an operation being called from within another operation,
if it will not be the case we will apply a reentrancy lock on the executeOp function.
```

5.5 Unclear Actions Setup

Design | Low | Version 1 | Acknowledged

CS-DMAv2II-020

There are several unclear code segments which do not have an impact on code correctness or security but are unclear:

1. Sometimes `paramsMap[1]` (or similar is used) when for `paramsMap[0]` is not even used. That makes the transaction simply more expensive. However, in the discussion with Summer.fi it was indicated that that is due to historical reasons.
2. Not always are important values written to transient storage. For example, the wrapping of ETH does not write how much ETH was wrapped. However, subsequent actions could potentially need that information (e.g. deposit the wrapped amount to Aave).

3. Further, it is often rather unclear what should be written to transient storage. For example, for Aave v3 deposits the amount of tokens deposited is written to transient storage and not the number of aTokens received. Note that due to rounding issues these two are not equal. Hence it could be helpful to get information on the data needed.
4. Sometimes there are some parts of actions that do not make much sense to be combined (e.g. adding transient storage values to the param makes sense, but the implementations also allow for doubling the param passed when transient storage is not read).
5. The SwapAction may receive input tokens back. These are not written to transient storage.

Note that in summary many actions are underspecified, inconsistent and unclear.

Acknowledged:

Summer.fi has acknowledged the above.

5.6 Unlimited Repayment on Behalf of Others

Design | Low | Version 1 | Risk Accepted

CS-DMAv2II-008

In the `AaveV3Payback` action, a loan can be repaid on behalf of another user. Further, the user can set the `paybackAll` flag, to repay an entire loan. However, combining these two features can lead to an unexpected increase in the amount of tokens repaid. If the beneficiary is aware of an incoming transaction that will repay their loan, they can frontrun the transaction to increase their debt. This would result in the caller repaying more tokens than they initially intended.

Note that the Aave specification specifies the following:

```
Use uint(-1) to repay the entire debt, ONLY when the repayment is not executed on behalf of a 3rd party.
```

Risk accepted:

Summer.fi accepts the risk and states:

```
We recognize the outlined risk and accept it in the current implementation.
```

Further, Summer.fi states the following future consideration:

```
We will take this issue into account for the next deployment of action contracts. Future versions will be
designed to mitigate this risk, possibly by incorporating additional checks or restrictions to prevent
frontrunning and ensure that repayments on behalf of third parties adhere to the specified guidelines.
```

6 Resolved Findings

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4

- Flashloan Action Stores Fees Only Code Corrected
- Array Out of Bounds Undetected Code Corrected
- Selector Retrieval For Small Data Code Corrected
- Transient Storage Is Not Cleared Code Corrected

Informational Findings	8

- Unused to When Borrowing Code Corrected
- Dead URL Code Corrected
- Explicit Visibility Code Corrected
- Floating Pragma Code Corrected
- Magic Values Should Be Constants Code Corrected
- Unused Imports Code Corrected
- Vulnerable Dependency Code Corrected
- Wrong Revert Message Emitted Code Corrected

6.1 Flashloan Action Stores Fees Only

Correctness Low Version 3 Code Corrected

CS-DMAv2II-022

The `TakeFlashloanBalancer` action stores the flashloan fees if `feePercentage > 0`, instead of storing the sum of fees and borrowed amount as required by the specification.

```
uint256 feePercentage = IProtocolFeesCollector(
  IVault(getRegisteredService(BALANCER_VAULT)).getProtocolFeesCollector()
).getFlashLoanFeePercentage();
if (feePercentage > 0) {
  uint256 product = amounts[0] * feePercentage;
  uint256 fullFlashloanAmount = product == 0 ? 0 : ((product - 1) / ONE) + 1;
  getTransactionStorageSlot().write(bytes32(fullFlashloanAmount));
} else {
  getTransactionStorageSlot().write(bytes32(amounts[0]));
}
```

There are no current actions reading the value from transient storages, so we consider this issue to be low severity.

Code corrected:

The code has been corrected.

6.2 Array Out of Bounds Undetected

Correctness Low Version 1 Code Corrected

CS-DMAv2II-002

The library `StorageSlot` implements arrays in transient storage. However, the `read*` functions do not implement bound checks on the accessed item in transient storage. Thus, it may be possible that `paramMapping - 1 > length - 1`. Ultimately, a zero item would be read. Nevertheless, it is not truly a stored item.

Code corrected:

The `read*` function now checks that the accessed item is within the bounds of the array in transient storage:

```
uint256 length = _getLength(slotPosition);

require(paramMapping <= length, "StorageSlot: Index out of bounds");
```

6.3 Selector Retrieval For Small Data

Correctness Low Version 1 Code Corrected

CS-DMAv2II-006

`isCallingAnExecutable()` takes `bytes` as an argument and tries to read the first `4` bytes to compare them to the desired selector. However, `convertBytesToBytes4()` does not handle arrays with a length of less than 4.

Code corrected:

The code has been adjusted to return `0x0` when the length is `< 4`. Thus, the `require` in `ActionAddress.execute` will fail in the cases described above.

6.4 Transient Storage Is Not Cleared

Design Low Version 1 Code Corrected

CS-DMAv2II-007

The `executeOp` function and the actions write to transient storage. While transient storage expires after a transaction, the transient storage is not cleared after `executeOp` has finished. Ultimately, calling

executeOp twice in the same transaction may lead to unintended consequences (e.g. call it through a multi(delegate)call).

Code corrected:

The code has been adjusted to set the length to 0. See Dirty Transient Storage for further consideration.

6.5 Wrong Revert Message Emitted

Informational | Version 3 | Code Corrected

CS-DMAv2II-021

The newly introduced flashloan reentrancy checker is implemented as follows

```
function checkIfFlashloanIsInProgress() private view {
  bytes4 errorSelector = FlashloanReentrancyAttempt.selector;
  bytes32 key = FLASHLOAN_LOCK_SLOT;
  assembly {
    let isFlashloanInProgress := tload(key)
    if eq(isFlashloanInProgress, 1) {
      mstore(0, errorSelector)
      revert(0x1c, 0x04)
    }
  }
}
```

Note that the revert message will be wrong. Namely, the `errorSelector` will be encode as `0x5fa25215000...0000`. Loading at `0x04` (4) at position `0x1c` (28) will thus return zero.

Code corrected:

The code has been corrected.

6.6 Dead URL

Informational | Version 1 | Code Corrected

CS-DMAv2II-009

The NatSpec of `executeOp()` points to the source of `AccountImplementation.sol`. However, the URL is dead.

Code corrected:

The implementation source code is reachable at the time of the fixes review.

6.7 Explicit Visibility

Informational | Version 1 | Code Corrected

CS-DMAv2II-010

The immutable variables in `OperationExecutor` have no visibility defined. While they are by default `public`, it is best practice to state the explicit visibility.

Similarly, `DIVISOR` in `CollectFee` has no visibility defined.

Code corrected:

The code has been corrected.

6.8 Floating Pragma

Informational | Version 1 | Code Corrected

CS-DMAv2II-011

Summer.fi uses a floating pragma solidity ^0.8.15. It is considered best practice to deploy contracts with the same compiler version and flags used in testing and audit to ensure that contracts are not inadvertently deployed with an outdated compiler version that could introduce bugs that negatively impact the contract system.

Code corrected:

Summer.fi has adjusted their codebase to use the fixed pragma `0.8.24` wherever suitable and meaningful.

6.9 Magic Values Should Be Constants

Informational | Version 1 | Code Corrected

CS-DMAv2II-014

The array names in storage are always typed out (e.g. `"actions"`). However, to prevent typographical errors in the future it could make sense to define them as constants. That would make the code more consistent as other strings are defined as constants.

Code corrected:

The code has been adjusted to use helper functions that use constants.

6.10 Unused Imports

Informational | Version 1 | Code Corrected

CS-DMAv2II-016

Many files have many unused imports. Below is an incomplete list of unused imports (note more exist and all files should be carefully evaluated):

1. OperationExecutor.sol: TakeFlashloan, Executable, IERC3156FlashLender
2. UseRegistry.sol: OPERATION_STORAGE
3. ProxyPermission.sol: FlashloanData, ServiceRegistry, DS_GUARD_FACTORY
4. SwapAction.sol: WETH, IWETH

5. CdpAllow.sol: OperationStorage, IVat, MathUtils, IWETH, WETH
6. Payback.sol: IVariableDebtToken, IWETHGateway, ILendingPool
7. Borrow.sol: IVariableDebtToken, IWETHGateway, ILendingPool
8. SetEMode.sol: IVariableDebtToken, IWETHGateway, ILendingPool
9. Withdraw.sol: OperationStorage
10. OpenVault.sol: OperationStorage

Code corrected:

Unused imports have been removed.

6.11 Unused `to` When Borrowing

Informational **Version 1** **Code Corrected**

CS-DMAv2II-017

The borrow action for Aave V3 has an unused `to` field in its data.

Code corrected:

The `to` has been removed.

6.12 Vulnerable Dependency

Informational **Version 1** **Code Corrected**

CS-DMAv2II-018

The Summer.fi repository imports the OpenZeppelin Contracts Library (Version 4.9.3) with a known vulnerability. More details can be found here: https://github.com/advisories/GHSA-9vx6-7xxf-x967

The contracts in scope do not use the vulnerable function, but it is considered best practice to upgrade to a patched version of the library.

Code corrected:

The Summer.fi repository has been updated to use the patched version of the OpenZeppelin Contracts Library (Version 4.9.6).

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 Gas Optimizations

Informational | Version 1 | Code Partially Corrected | Acknowledged

CS-DMAv2II-012

Below, some potential gas-saving opportunities are listed that could reduce gas consumption by some degree to reduce execution cost:

1. `ActionAddress`: The `callData` bytes could save 4 bytes of call data per call if the input to the `executeOp` calls does not include the `execute` selector. `ActionAddress.execute` could encode the arguments with the selector. Ultimately `4x` bytes of calldata could be saved where `x` is the number of calls. That may especially be relevant for saving gas on L2s.
2. `isFlashloanInProgress` is a storage variable to detect flashloans and is typically set in the context of the `OperationExecutor`. Given the transient nature of the variable, it could be stored in transient storage.
3. Gas overhead with ServiceRegistry: It could be possible to replace the ServiceRegistry with a version that batches calls. Ultimately, that could reduce the number of `staticcall` calls to 1 in the loop in `aggregate()`.

Further, below are some further opportunities to remove some redundant operations (not necessarily gas savings):

1. `isCallingAnExecutable()`: encodes the `execute` selector and loads the first four bytes. However, this operation is redundant as the `.selector` returns 4 bytes as desired.
2. `getStorageSlotPosition`: casts to a `bytes32` to `uint256` and back to `bytes32`. The operation is redundant.
3. OperatorExecutor: The visibility of `executeOp` can be restricted to *external* instead of *public*.
4. `FixedPoint` / `SafeMath`: The overflow/underflow checks are redundant as solidity version ^0.8.0 enforces them by default.

Code corrected:

The gas optimization 2 has been implemented. The redundant operations 1, 2, and 3 have been removed from the code.

Code partially corrected:

`SafeMath` is still used in actions but has been replaced in some places.

Acknowledged:

The gas optimization 1 and 3 have been acknowledged. Summer.fi also acknowledgeds the partially corrected remaining issue.

7.2 Lack of Ownership Transfer Events

Informational | Version 1 | Acknowledged

CS-DMAv2II-013

`OperationsRegistry.transferOwnership()` does not emit an event. Events can help off-chain applications retrieve data and should at least be emitted for important actions.

Acknowledged:

Summer.fi acknowledged the issue and replied:

```
In the near future we are planning to refactor the OperationRegistry and potentially not
use it in current form
```

7.3 Unused Contracts

Informational | Version 1 | Acknowledged

CS-DMAv2II-015

The contract `UseStore` is not used anywhere.

Further, natspec comments referencing `UseStore` are outdated:

- aave/v2/Deposit.sol
- aave/v2/Payback.sol
- aave/v3/Deposit.sol
- aave/v3/Payback.sol
- aave/v3l2/Deposit.sol
- aave/v3l2/Payback.sol
- common/SetApproval.sol
- common/UnwrapEth.sol
- common/UseStore.sol
- common/WrapEth.sol
- morpho-blue/Borrow.sol
- morpho-blue/Deposit.sol
- morpho-blue/Payback.sol
- spark/Deposit.sol
- spark/Payback.sol

Acknowledged:

Summer.fi acknowledges the issue and stated:

```
This repository holds some contract that are not used in it’s context, but are used in
summerfi-monorepo or `oasis-earn-sc` - we are planning to merge all repositories and only
use summerfi-monorepo before the next deployment of core contracts
```

7.4 WrapEth Functionality

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

CS-DMAv2II-019

The WrapEth functionality could be generalized to wrap native tokens (e.g. wrapping matic to Wrapped Matic on Polygon). However, the naming and the hardcoded "WETH" string for querying the service registry may make it unsuitable for other native tokens.

Acknowledged:

Summer.fi stated:

```
We are not planning deployments on additional chains for now. However, we will make the
necessary changes if we decide to proceed with it.
```

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 Dirty Transient Storage

Note Version 1

The fix for Transient storage is not cleared introduced that transient storage is cleared. However, that is not fully the case. Namely, only the length of the transient storage arrays is set to zero and transient storage could be dirty. However, as long as the transient storage is only written with `StorageSlot.write()` and read by `StorageSlot.read()` the code will work as intended.

8.2 Limitations on Correctness

Note Version 1

Users and governance should be aware that individuals could break the intended behavior of the system. For example, a user could execute independent manipulations of transient storage at the given locations so that the execution could be not as intended by governance and developers (e.g. clear transient storage locations, add hashes to the array to bypass check for the correctness of operation hashes). However, that is limited to the proxy of the user. Moreover, the intended functionality of the design could be bypassed.

8.3 No Funds in Executor

Note Version 1

The users may send funds to the executor to repay flashloan debt. Users should validate that they do not send too many funds to the executor, as these could be stolen.

8.4 No Permissions Should Be Held by the Executor

Note Version 1

Users should be aware that the `OperationExecutor` should never be given any permissions (except for the duration of a flashloan where they are required to enable executions in the context of proxy). If the executor holds any permission, the user may be vulnerable to attacks (e.g. `receiveFlashloan()` could be used to call into the proxy of the user).

8.5 Registry Usage

Note Version 1

Note that the service registry is sometimes not used for certain cases:

1. `MCD_FLASH`: The service registry is immutable. However, the flashloan module has been subject to changes. Thus, MakerDAO's chainlog is used to allow updates of `MCD_FLASH`.
2. The balancer vault is defined in the service registry and used in some places. However, in the `receiveFlashloan` callback, a constant is used. Summer.fi elaborated that this is most likely due to historical reasons.

8.6 Supported Proxies

Note **Version 1**

While in theory any proxy can delegatecall the executor, some functionality may be unsupported by some proxies. Namely, to support flashloan actions, the proxy must satisfy the following:

1. be a DPM proxy
2. be a DSProxy with either no authority contract or an authority contract that supports the full interface of the guards used by DPM proxies
3. be an arbitrary proxy that supports one of the interfaces implemented by the two above (execution, access control, ...)

If not, the permission handling will revert and flashloan functionality will not work. Further, note that operations with other contracts may lead to some storage collisions or similar. Thus, 3 should always be carefully evaluated for suitability.

8.7 Use of `address.transfer`

Note **Version 1**

Using `address.transfer` has been discouraged due to the possibility of changing gas costs of opcodes. Further, some L2s (e.g. zkSync) do not support such transfers with low gas due to their internal mechanics. Ultimately, Summer.fi should be aware that transferring ETH in such a way may not be future proof.