BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Defi Money

Fee Module

September 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	DEFI MONEY - FEE MODULE
Website	defi.money
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/defidotmoney/dfm-contracts/tree/58e83d59544f57af2719295e1d025a6b032b6c08/contracts/fees
Resolution 1	https://github.com/defidotmoney/dfm-contracts/tree/56907e2b6457bd3d91c2003065e5f233bf82cb97/contracts/fees

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	4	3		1
Medium	5	2		3
Low	18	3		15
Informational	8	3		5
Governance	3			3
Total	38	11		27

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Fees/dependencies

FeeConverterBase

The FeeConverterBase is an abstract contract which is inherited by the FeeConverter and FeeConverterWithBridge contracts. This contract allows users to provide the stableCoin in exchange to one of the collateral tokens or to provide native ETH in exchange to the stableCoin token. This contract receives fees in form of the different collateral tokens from the main protocol.

For both conversion processes, a bonus is applied on the provided input amount which results in an increased output amount for users. The stablecoin value is assumed to be 1 USD.

This has the following ideas behind it:

a) Stablecoin -> Collateral: Whenever the stableCoin depegs, this swap path gives users the possibility to not only prevent losses but also experience a gain. In fact as long as the stableCoin is not above the peg, this trade will always be favorable for users. In reverse, this means that users need to purchase the stableCoin on all markets (CEX; DEX; OTC) which creates buying pressure and then helps to reach back to the peg.
The rationale behind this is to ensure that the contract has always a sufficient balance of the stableCoin plus it additionally serves as peg-restoring mechanism.

b) ETH -> Stablecoin: Whenever the stableCoin is above the peg, this trade will give users the possibility to purchase the stableCoin at a discount while providing ETH to the relayer. If the stableCoin stays at 1 USD, it will still be favorable for users to execute this swap due to the applied bonus.
The rationale behind this is to ensure that the relayer always has sufficient ETH, which is simultaneously also the limitation for this conversion: If the relayer has sufficient ETH, this conversion is not allowed.

Since the likelihood of a depeg is higher than that of the stableCoin being traded above the peg plus the additional fact that a bonus is applied upon Stablecoin -> Collateral swaps, users are most of the time incentivized to swap the stableCoin for the existing collateral. This step is essential for the functioning of the protocol because it is assumed/desired that the fee module will always have sufficient stablecoins inside in order to be able to execute its logic.

Privileged Functions

- setIsEnabled
- setPrimaryChainFeeAggregator
- setSwapBonusPctBps
- setSwapMaxBonusAmount
- setRelayMinBalance
- setRelayMaxSwapDebtAmount

Issue_01	stableCoin is drainable if it has a corresponding oracle value
Severity	**High**
Description	The swapDebtForColl function allows users to provide the stableCoin and receive a collateral token as outputToken. In the scenario where the stableCoin address is provided as outputToken AND the following call returns a valid price: *mainController.get_oracle_price(address(outputToken));* This allows users to drain the stableCoin token from the contract by providing for example 100e18 stableCoin tokens as amountIn. Furthermore, we assume a price of 1e18. Due to the fact that a bonus is applied on the provided stableCoin amount, this can be abused to drain stableCoin tokens from the contract. Consider a 10% bonus: -> debtAmountIn + bonus = 110e18 -> user receives 110e18 stableCoin tokens while having only provided 100e18 stableCoin tokens This issue is amplified if the stableCoin token is below peg and the get_oracle_price function returns a smaller price.
Recommendations	Consider ensuring that outputToken can never be the stableCoin address.
Comments / Resolution	Resolved, a check has been implemented which prevents outputToken from being the stableCoin.

Issue_02	swapMaxBonusAmount safeguard can be trivially bypassed via multi swaps
Severity	**Medium**
Description	The swapDebtForColl function allows users to swap their stableCoin tokens for any existing collateral within the contract. There are a few additional points to mention first: a) 1e18 of stableCoin is always valued at 1 USD b) Every swap will apply a bonus which is artificially added towards the provided stableCoin value The rationale of this function is to create buy pressure for the stableCoin on markets (CEX; DEX; OTC) to purchase the stableCoin and then sell the stableCoin via this function. This is specifically useful in the event of a depeg as that would mean the buy pressure will help the stableCoin to repeg. A limitation in how much bonus can be received was introduced as follows: *if (bonus > swapMaxBonusAmount) bonus = swapMaxBonusAmount;* This limitation can be trivially bypassed by executing multiple consecutive calls.
Recommendations	The development of a more sophisticated implementation which also includes sybil prevention etc. is very intrusive. Therefore, a simple time-management limitation could be implemented (which will of course still have its own downsides).
Comments / Resolution	Resolved, this check has been completely removed. It is now possible to swap without a limit.

Issue_03	Sole token validation via oracle *can be* insufficient
Severity	**Low**
Description	Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic. This partially applies to swap functions, as users can provide an arbitrary input parameter for outputToken via the swapDebtForColl function. The only validation is non-explicit and happens via: *uint256 price = mainController.get_oracle_price(address(outputToken));* This means that users can eventually directly transfer tokens to the contract and then invoke the swapDebtForColl function. While for now it is clear that there is no downside to the protocol, this still falls user "unnecessary user flexibility" which *can potentially* result in unforeseen edge-cases.
Recommendations	Consider implementing a simple whitelist mechanism to explicitly validate the outputToken parameter within the swapDebtForColl function.
Comments / Resolution	Acknowledged.

Issue_04	Extended time above peg will result in malfunction of fee module
Severity	**Low**
Description	The fee module inherently assumes that the contract mostly contains the stableCoin in an effort to work properly (distribute fees,...) If the stableCoin is significantly above the peg for an extended time period this means that users will not be incentivized to convert their stableCoin to collateral tokens via the swapDebtForColl function. This in turn means the contract will not receive any stableCoin tokens and therefore the fee module will not work as desired.
Recommendations	Consider increasing the bonus appropriately if the stableCoin is above the peg for an extended duration.
Comments / Resolution	Resolved, this will be handled manually by governance.

Issue_05	Lack of address(0) check during _setPrimaryChainFeeAggregator()
Severity	**Low**
Description	The _setPrimaryChainFeeAggregator function allows for the setting of primaryChainFeeAggregator. Currently, there is no validation against address(0), which means in the scenario where primaryChainFeeAggregator becomes address(0, fee transfers will revert.
Recommendations	Consider thinking about if this can become an issue or is desired practice to prevent fee transfers in certain emergency scenarios.
Comments / Resolution	Acknowledged.

Issue_06	swapNativeForDebt function may result in relayer being excessively overfunded
Severity	**Informational**
Description	The swapNativeForDebt function allows users to provide native ETH in an effort to receive the stableCoin. This function can be called as long as the relayer is underfunded and has an upper limit of relayMaxSwapDebtAmount which represents the upper stableCoin amount which can be received per swap. Users may slightly trick this function by executing two swaps: a) Providing an ETH amount which results in the relayer balance becoming relayMinBalance -1 b) Providing an ETH amount which results in receiving exactly the relayMaxSwapDebtAmount of stableCoin tokens.
Recommendations	Consider if this can become a problem or if it is a design choice. In the latter scenario no changes should be made.
Comments / Resolution	Acknowledged.

Issue_07	Lack of decimal normalization during getSwapNativeForDebtAmountOut can result in incorrect conversion
Severity	**Informational**
Description	The getSwapNativeForDebtAmountOut function calculates how much of stableCoin is being received based on the provided nativeAmountIn. Additionally, it applies a bonus on top of nativeAmountIn such that the user will receive more from the stableCoin. The conversion is as follows: > nativeAmountIn + bonus * price / 1e18 This has the following assumptions: a) price is returned with 18 decimals b) stableCoin is denominated with 18 decimals For this protocol, all assumptions are accurate. However, if this protocol is forked and these assumptions are not guaranteed anymore, the conversion will be broken.
Recommendations	Following the path of least resistance, we do not recommend a change. However, ideally the calculation would adjust for different decimals.
Comments / Resolution	Resolved, this is just a note for potential forkers to consider.

GaugeAllocReceiverBase

The GaugeAllocReceiverBase contract is an abstract contract which is inherited by the ReceiverVoteMarket and ReceiverVotium contracts.

This contract serves as a registry-like contract where the owner can craft and modify a list of gauges with corresponding weights by leveraging OpenZeppelin's enumerableSet library. These gauges are then used within the ReceiverVoteMarket and ReceiverVotium contracts to allocate incentives based on the overall amount of fees (stableCoin) in these contracts.

Furthermore, it ensures that the maximum approval to the corresponding external contract (Votium, VoteMarket) is granted upon deployment.

Privileged Functions

- setGauges

No issues found.

LocalReceiverBase

The LocalReceiverBase is an abstract contract which is inherited by the LzComposeForwarder and ReceiverVoteMarket contracts. These contracts are only deployed on the primary chain.

It exposes the basic functionality of fee notification as the notifyNewFees function which is solely invoked by the PrimaryFeeAggregator contract during the weekly fee procession. The only use of this contract is to expose a shared receiver base which can be inherited.

Privileged Functions

- none

No issues found.

LzComposeReceiverBase

The LzComposeReceiverBase is an abstract contract which is exclusively inherited by the ReceiverVotium contract on the non-primary chain.

This contract contains logic to comply with LayerZero's message compose feature after the stableCoin has been transferred from the PrimaryFeeAggregator on the primary chain to the Votium Module on the non-primary chain.

More information on the compose message flow can be found under: "Appendix: Cross-Chain Compose Message"

Privileged Functions

- none

No issues found.

TokenRecovery

The TokenRecovery contract is a simple abstract contract that facilitates the approval management and withdrawals of tokens. It is inherited by the following contracts:

1) FeeConverterBase (FeeConverter; FeeConverterWithBridge)
2) GaugeAllocReceiverBase (ReceiverVoteMarket, ReceiverVotium)
3) LzComposeForwarder
4) PrimaryFeeAggregator

The owner can withdraw any token via the transferToken function as well as granting approvals for any token to any recipient via the setTokenApproval function.

Privileged Functions

- transferTokens
- setTokenApprovals

Issue_08	Governance Privilege: Owner can withdraw funds
Severity	**Governance**
Description	Any contract which inherits this abstract contract exposes functionalities for the owner to withdraw all funds. This will become an issue if governance is compromised or in other similar scenarios.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_09	Lack of native token withdrawals
Severity	**Low**
Description	Since this module allows for withdrawing ERC20 tokens, it may also be desired to withdraw the native token in some edge-scenarios where users have accidentally directly transferred the native token to the FeeConverterWithBridge or PrimaryFeeAggregator contracts. This is currently not possible because the contract does not expose such a functionality.
Recommendations	Consider implementing a function which allows for the withdrawal of the native token.
Comments / Resolution	Acknowledged.

Fees

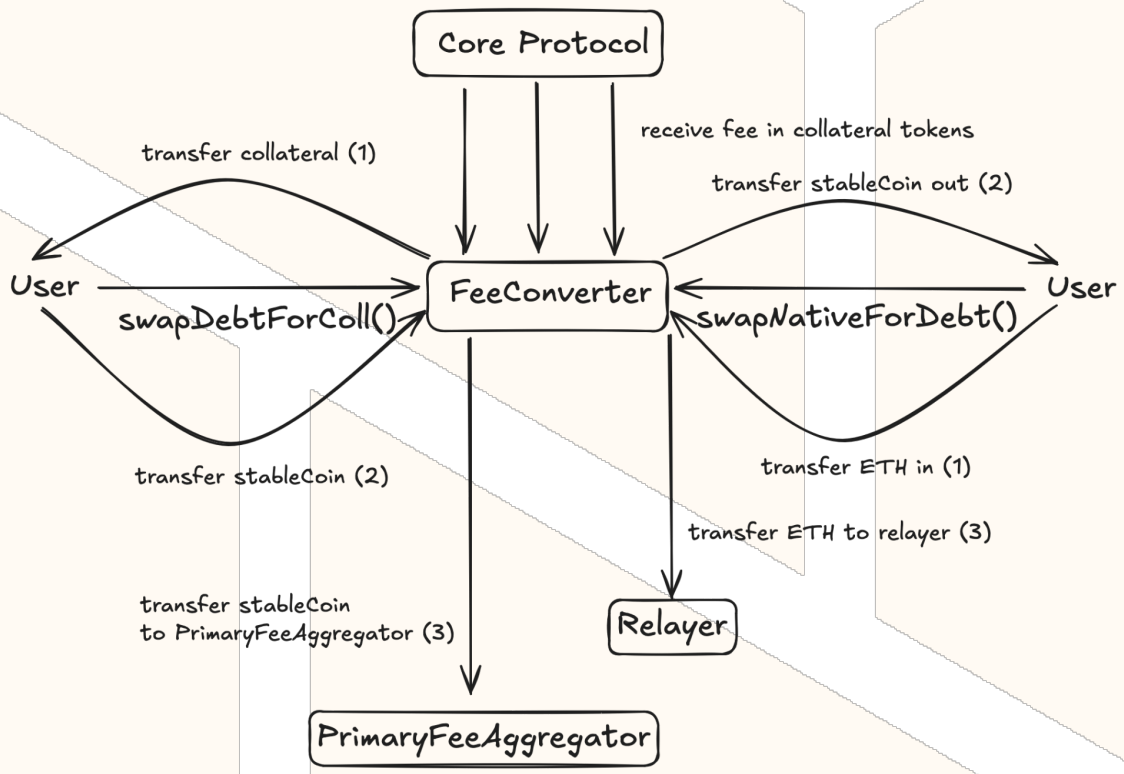
FeeConverter

The FeeConverter contract inherits the FeeConverterBase contract and extends its functionality by overriding the swapDebtForColl function to include an automatic transfer of all stableCoin tokens in the contract to the PrimaryFeeAggregator where these fees will then be distributed for different purposes.

Furthermore, this contract exposes a permissionless transferToAggregator function that allows anyone to transfer all stableCoin tokens within this contract to the primaryChainFeeAggregator.

This contract is deployed on Optimism which is corresponding to the primary chain.

Appendix: Fee Accumulation and Flow (Primary Chain)



Privileged Functions

- transferToken
- setTokenApproval
- setIsEnabled
- setPrimaryChainFeeAggregator
- setSwapBonusPctBps
- setSwapMaxBonusAmount
- setRelayMinBalance
- setRelayMaxSwapDebtAmount

Issue_10	Griefing: Malicious user can permanently keep relayer underfunded
Severity	**High**
Description	The function swapNativeForDebt is fundamental within the contract and ensures that the relayer is permanently and sufficiently funded with ETH to function properly. A problem arises due to the fact that the transferToAggregator function is permissionless, thus a malicious user can invoke this function which then automatically transfers the whole balance of stableCoin out: *stableCoin.transfer(primaryChainFeeAggregator, stableCoin.balanceOf(address(this)));* If the contract has an empty stableCoin balance, the swapNativeForDebt function constantly reverts which then results in the relayer being permanently under-funded. Furthermore, if the relayer is permanently underfunded that means that the swapDebtForColl function can never be invoked which would then result in a negative feedback loop because the contract ends with only collateral inside while no one can get this collateral out:

	require(!canSwapNativeForDebt(), "DFM: swapNativeForDebt first");
Recommendations	In the current implementation, the best solution is to mark the transferToAggregator function as internal because it is then only invoked upon the swapDebtForColl function which beforehand ensures that the relayer is sufficiently funded. However, if the issue below is considered, this recommendation will not work anymore. Whatever solution is chosen, it must be ensured that the stableCoin balance can never be transferred out in the scenario where the relayer is insufficiently funded.
Comments / Resolution	Resolved, the _transferToAggregator function has been marked as internal and an external transferToAggregator function has been implemented which can be disabled to prevent such abusive behavior plus is only be callable when the relayer is sufficiently funded.

Issue_11	swapDebtForColl does automatically transfer out all stableCoin tokens, rendering the contract empty for swapNativeForDebt calls
Severity	**High**
Description	The only possibility of how the contract accumulates the stableCoin tokens is via the swapDebtForColl function which allows users to provide stableCoin tokens in exchange for collateral tokens. At the end of the swapDebtForColl function, all stableCoin tokens are being transferred out to the primaryChainFeeAggregator contract: *transferToAggregator();* This means, the contract is always in a state where there are no funds left to transfer out during the swapNativeForDebt function. While it is clear that the swapDebtForColl function can only be called if the relayer is sufficiently funded, at some point the relayer will become underfunded and then it is necessary to invoke swapNativeForDebt, which will however not work because there are no stableCoin tokens within the contract. This state is permanently existent besides someone manually transfers some stableCoin tokens to the contract.
Recommendations	Consider keeping a small threshold of stableCoin tokens inside the contract to ensure that the swapNativeForDebt function can be invoked.
Comments / Resolution	Acknowledged.

Issue_12	Mandatory canSwapNativeForDebt check within swapDebtForColl can result in desired fee module state never being reached
Severity	**Medium**
Description	The contract aims to transfer out collateral tokens and attract the stableCoin token in an effort to work properly. This only works via the swapDebtForColl function. The swapDebtForColl function can however only be invoked if the relayer is sufficiently funded to execute certain actions. If the stableCoin is too heavily depegged, users will never invoke the swapNativeForDebt function because they will essentially lose money with this trade, thus, without governance intervention, the relayer will not receive any funds. This means that the protocol is in a state where it urgently incentivizes users to buy the stableCoin on the market to restore the peg which means users will attempt to trade in the stableCoin via the swapDebtForColl function, which however never works because the relayer will not be funded due to a lack of incentives for users to invoke swapNativeForDebt (due to stableCoin depeg).
Recommendations	Consider re-thinking this design choice.
Comments / Resolution	Acknowledged.

Issue_13	transferToAggregator is not guarded with whenEnabled modifier
Severity	**Low**
Description	The transferToAggregator function remains unguarded which is inconsistent in the broader contract context, as all functions are guarded with the whenEnabled modifier.
Recommendations	Consider guarding this function accordingly.
Comments / Resolution	Resolved.

Issue_14	Unused variables
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. *uint16 public bridgeBonusPctBps;* *uint256 public maxBridgeBonusAmount;*
Recommendations	Consider removing these unused variables.
Comments / Resolution	Resolved.

FeeConverterWithBridge

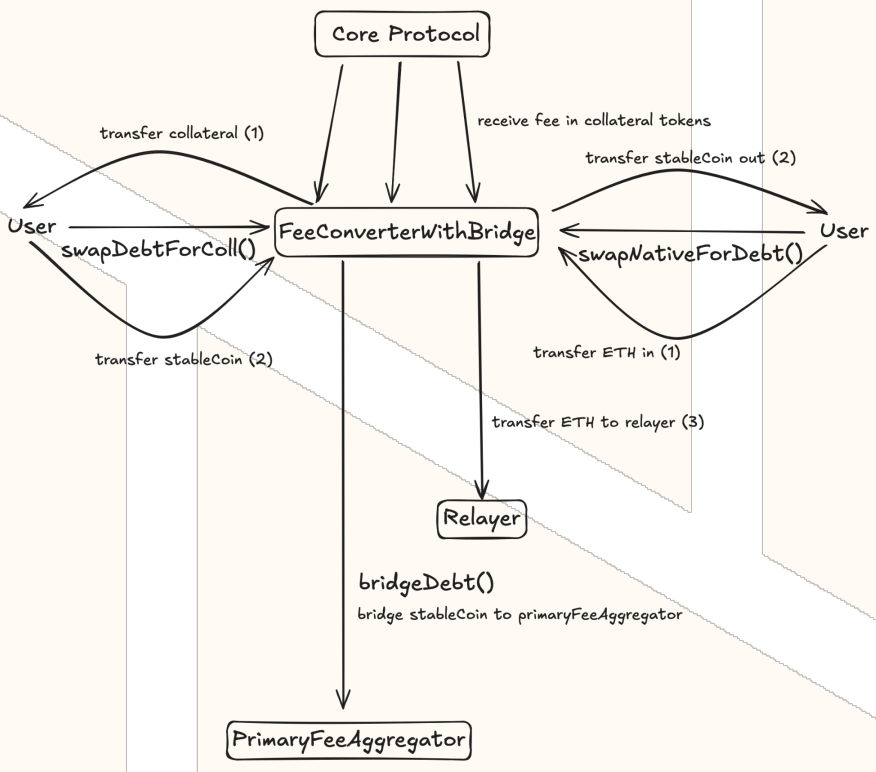
The FeeConverterWithBridge contract inherits the FeeConverterBase and extends its functionality with a cross-chain transfer of the stableCoin token to the primaryFeeAggregator contract on the primary chain.

The bridgeDebt function is permissionless and can be invoked whenever the relayer is sufficiently refunded. During this bridge call, all stableCoin tokens in the contract will be transferred out and the caller will receive a small reward as incentive to do so.

The reward is calculated based on the bridgeBonusPctBps variable and the total amount of stableCoin tokens within this contract.

An additional safeguard was implemented which limits the maximum bonus amount a user will receive per bridgeDebt call and is defined as bridgeMaxBonusAmount.

Appendix: Fee Accumulation and Flow (Secondary Chain)



Appendix: Simple Bridge

Below we will illustrate the simple bridge flow when stableCoin tokens are transferred from secondary chains to the primary chain:

> FeeConverterWithBridge.bridgeDebt

> OFT.sendSimple(primaryId, receiver)

> OFT.sendSimple(_eid, _target, _amount)

-> _debit(_amountLd, _minAmountLd, _dstEid)

-> _debitView(_amountLd, _minAmountLd, _dstEid)

-> _burn(msg.sender, amountSendLd)

-> encode(_sendTo, _amountShared, _composeMsg)

-> _lzSend(_eid, message, options, MessagingFee)

-> _payNative(fee.nativeFee)

-> Endpoint.send(MessagingParams, _refundAddress)

> Endpoint.send(MessagingParams, _refundAddress)

-> _send(msg.sender, _params)

-> _outbound(_sender, dstEid, receiver)

-> getSendLibrary(_sender, dstEid)

-> SendLibrary.send

-> _payNative(nativeFee, suppliedNative, _sendLibrary, _refundAddress)

> OFFCHAIN

> Endpoint.verify(Origin, _receiver, _payloadHash)

-> _inbound(_origin, _receiver, _payloadHash)

> Endpoint.lzReceive(_origin, _receiver, _guid, _message, _extraData)

-> _clearPayload(_receiver, _srcEid, _sender, _nonce, _payload)

-> OFT.lzReceive(_origin, _guid, _message, msg.sender, _extraData)

> OFT.lzReceive(_origin, _guid, _message, _executor, _extraData)

-> _lzReceive(_origin, _guid, _message, _executor, _extraData)

-> _credit(toAddress, amountLD, srcEid)

-> _mint(_to, _amountLD)

Privileged Functions

- transferToken
- setTokenApproval
- setIsEnabled
- setPrimaryChainFeeAggregator
- setSwapBonusPctBps
- setSwapMaxBonusAmount
- setRelayMinBalance
- setRelayMaxSwapDebtAmount
- setBridgeBonusPctBps
- setBridgeMaxBonusAmount

Issue_15	Bridging will not work if stableCoin is depegged
Severity	**Low**
Description	The bridgeDebt function allows for bridging the stableCoin token from the non-primary chain to the primary chain's primaryChainFeeAggregator address. This will only be allowed if the relayer on the non-primary chain is sufficiently funded: *require(!canSwapNativeForDebt(), "DFM: swapNativeForDebt first");* This check is accurate and reasonable as it prevents users from bridging stableCoin tokens while the relayer is underfunded (which would mean the contract has no stableCoin tokens left to exchange for native ETH). However, at the same time, this methodology implements an edge-case. If the stableCoin token is depegged, this means that users will not invoke the swapNativeForDebt function because this essentially results in a loss (depending on the depeg intensity) In such a scenario, the relayer may become underfunded and bridging would not be allowed until the stableCoin token is sufficiently pegged again to be a net-negative trade for users to invoke the swapNativeForDebt function (or someone transfers manually ETH to the relayer).
Recommendations	We do not recommend a change at this point. However, this situation should be carefully considered and monitored for.
Comments / Resolution	Acknowledged.

Issue_16	bridgeMaxBonusAmount safeguard can be bypassed using multiple subsequent calls
Severity	**Low**
Description	The reward which is received upon each bridgeDebt call is safeguarded with the bridgeMaxBonusAmount variable. If the reward exceeds this amount, it will simply be downsized to it: *debtReward = (amount * bridgeBonusPctBps) / MAX_BPS;* *if (debtReward > bridgeMaxBonusAmount) debtReward = bridgeMaxBonusAmount;* *return debtReward;* This can be trivially circumvented by calling the bridgeDebt function with a higher frequency. This issue was only rated as low severity because users cannot input the desired amount but the amount is rather determined by the contract balance.
Recommendations	Consider if that becomes a problem, if yes consider implementing time-based limitations on how often this function can be called per user (even though that will not guard against sybil attacks).
Comments / Resolution	Acknowledged.

Issue_17	Lack of upper limit for bridgeBonusPctBps during constructor
Severity	**Low**
Description	The constructor sets bridgeBonusPctBps without any upper validation, this could result in this value becoming > 10_000, effectively breaking the contract.
Recommendations	Consider executing the same check as within setBridgeBonusPctBps.
Comments / Resolution	Resolved.

LzComposeForwarder

The LzComposeForwarder contract is deployed on the primary chain and inherits the LocalReceiverBase. The goal of this contract is to receive the stableCoin from the PrimaryFeeAggregator in the same fashion as other fee modules receive it, via the notifyNewFees function. Once fees are received and this function is invoked there are two possible scenarios:

1. Every x weeks: Bridge the stableCoin with a compose message to the secondary chain and invoke the ReceiverVotium contract's _notifyNewFees function (via lzCompose) to deposit funds into the Votium contract.

2. Every "non-x" weeks: Receive the stableCoin and only accumulate it

These scenarios solely depend on the bridgeEpochFrequency variable, as this is the frequency of how often tokens are bridged (in weeks).

The cross-chain flow can be found under "Appendix: Cross-Chain Compose Message"

Privileged Functions

- transferToken
- setTokenApproval

Issue_18	quoteNotifyNewFees function may return incorrect quote due to logical blunder
Severity	**Medium**
Description	The quoteNotifyNewFees function is invoked during PrimaryFeeAggregator.quoteProcessWeeklyDistribution to determine the fee which must be provided as msg.value. Due to the fact that the at the time of the quote, no tokens have been transferred into the contract yet, this may return 0: *uint256 amount = stableCoin.balanceOf(address(this)); if (amount >= MIN_AMOUNT) {* *...* *}* Which then in turn results in an incorrect quote for the weekly distribution.
Recommendations	Consider incorporating the “to be transferred” balance into this function.
Comments / Resolution	Resolved.

Issue_19	Immutability of bridgeEpochFrequency will significantly reduce flexibility
Severity	**Low**
Description	The _notifyNewFees function only triggers bridging of the stableCoin every X weeks, this is determined via the bridgeEpochFrequency variable and the following condition: *if (getWeek() % bridgeEpochFrequency == 0)* There are two downsides of this implementation: a) If the _notifyNewFees function is not triggered during the target week, one needs to wait another x weeks until it can be triggered (unlikely to happen due to permissionless nature) b) In any future scenarios, the frequency can never be changed. This may become a significant problem if the initial frequence was for example 4 weeks and the protocol grows significantly which now requires a more frequent bridging approach.
Recommendations	Consider removing the immutability from the bridgeEpochFrequency variable and introduce a setter function (onlyOwner).
Comments / Resolution	Acknowledged.

PrimaryFeeAggregator

The PrimaryFeeAggregator is the main contract which aggregates all fees. It receives fees indirectly from the main protocol via the FeeConverter and FeeConverterWithBridge.

The contract owner can determine different fee receivers and corresponding weights which are sectioned in priorityReceiver(s) and one fallbackReceiver. In the current development state these are namely the following:

a) ReceiverVoteMarket (priorityReceiver)
b) ReceiverVotium (priorityReceiver)
c) StableStaker (fallbackReceiver)

All addresses within the priorityReceivers array will have an assigned pctInBps weight whereas the overall share of tokens for priorityReceivers is determined by all aggregated pctInBps variables into the totalPriorityReceiverPct value. Furthermore, each priorityReceiver has a corresponding maximumAmount which serves as the upper limit on each distribution iteration.

Once receivers are set, anyone can invoke the processWeeklyDistribution function once per week to distribute any aggregated fees among all recipients. The distribution is happening using the old-fashioned loop technique where it first loops over all priorityReceivers and distributes the total balance on their corresponding weights. Any remaining leftover balance is then afterwards distributed to the fallbackReceiver address.

In an effort to incentivize users to execute the fee distribution, users will receive a fixed amount of the stableCoin token as refund, which is determined as callerIncentive.

Privileged Functions

- transferToken
- setTokenApproval
- addPriorityReceivers
- removePriorityReceiver

- setFallbackReceiver
- setCallerIncentive

Issue_20	Lack of address(0) check during setFallbackReceiver function
Severity	**Low**
Description	The setFallbackReceiver function allows for the setting of the fallbackReceiver address . Currently, there is no validation against address(0), which means in the scenario where fallBackReceiver becomes address(0), the processWeeklyDistribution function reverts.
Recommendations	Consider implementing the aforementioned check.
Comments / Resolution	Acknowledged.

Issue_21	Lack of priorityReceiver duplication check
Severity	**Informational**
Description	The addPriorityReceivers function allows the owner to add addresses to the priorityReceivers array. Currently there is no check which avoids duplicate addresses in the array.
Recommendations	This is a best-practice recommendation and does not expose any risk in the current iteration. Therefore we do not recommend a fix to ensure the codebase does not experience any change post-audit. For future iterations, we recommend simply switching to an enumerableSet.
Comments / Resolution	Acknowledged.

ReceiverVoteMarket

The ReceiverVoteMarket contract inherits LocalReceiverBase and GaugeReceiverBase contracts in an effort to build a full module for accepting fees and keeping track of assigned gauges.

Funds are received from the PrimaryFeeAggregator during the processWeeklyDistribution call in the form of the stableCoin token. Whenever the _notifyNewFees function is invoked it will first calculate the corresponding amount for each gauge based on the weight and then either create a new bounty or extend an existing bounty in the Platform contract.

The contract does furthermore expose a list of excluded addresses which can be modified by the contract owner. This list is then used upon bounty creation to prevent these addresses from participating in the bounty.

Appendix: Platform Usage

The Platform contract is the external contract which is invoked by the ReceiverVoteMarket contract upon bounty creation and bounty extension.

Using this contract, anyone can create bounties for specific gauges and users can claim rewards for a bounty period by providing a Merkle proof of their gauge voting record. The contract verifies the proof against a trusted snapshot, calculates the user's share of rewards based on their voting power, and transfers tokens to them.

Additionally to the pure creation of a bounty, an already existing bounty can be increased in duration and amount which then takes effect from the next period. This functionality is also used within the ReceiverVoteMarket contract.

Furthermore, this contract exposes a permissionless closeBounty function, which refunds any remaining funds from a bounty to the original manager contract which is the ReceiverVoteMarket in our scenario.

Privileged Functions

- transferToken

- setTokenApproval
- setExclusionList
- setGauges

Issue_22	getExclusionList() is rendered useless because bounty extension does not adhere to eventual modifications
Severity	**Medium**
Description	The contract exposes an exclusion list where the owner can add/remove addresses to/from. This is then used upon new bounty creations and prevents certain addresses from participating in the bounty. A problem arises if a bounty is extended and new addresses are added to the list. This would mean any newly added address should not be able to participate in bounties created by this contract. (The same goes for removed addresses) However, if a bounty for a gauge is steadily increased before it has expired, it will never reflect the newly added addresses in the exclusion list, effectively rendering this feature useless (specifically because the fee distribution is permissionless, users can always trigger the extension of a bounty each week, while the bounty period is 4 weeks).
Recommendations	Consider never extending bounties if the exclusion list has been modified. In that scenario it should always create a new bounty. Optionally, one could simply remove the bounty extension logic and create a new bounty every week.
Comments / Resolution	Acknowledged.

ReceiverVotium

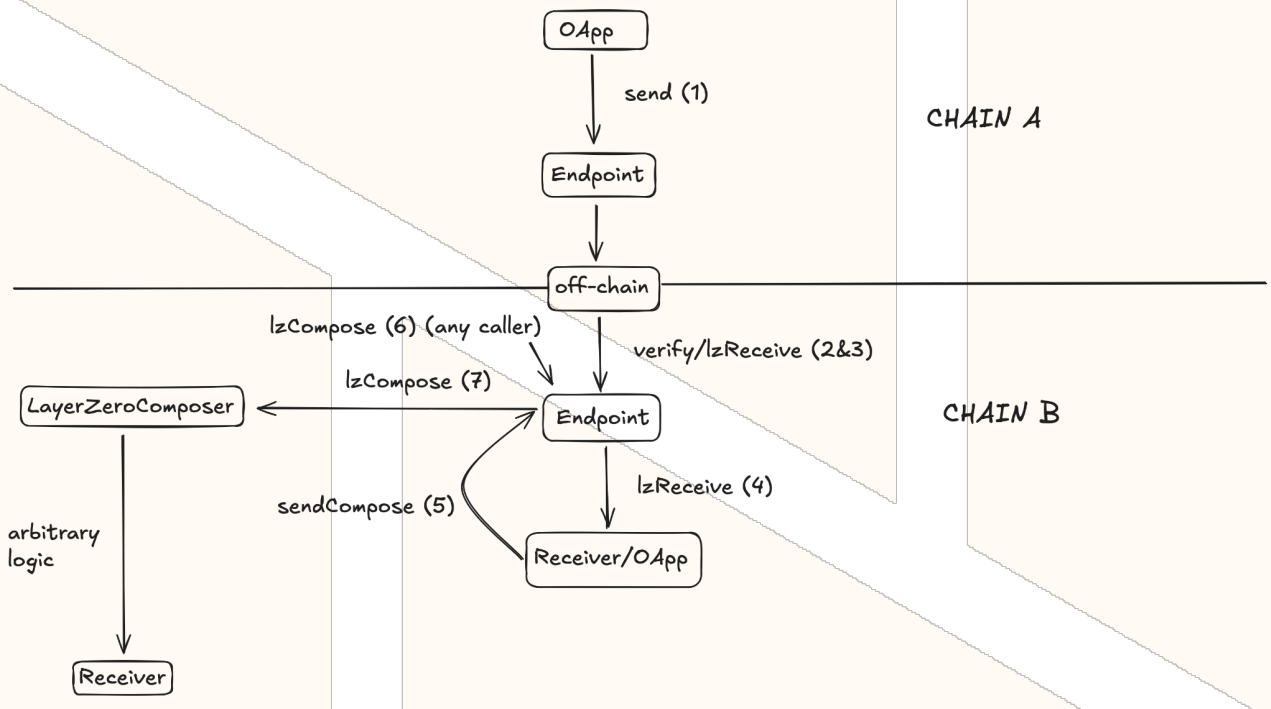
The ReceiverVotium contract inherits the GaugeAllocReceiverBase and LzComposeReceiverBase contracts in an effort to incorporate a list of gauges and to be able to receive compose messages from the LzComposeForwarder contract on Optimism.

The main logic of this contract is to receive fees in the form of the stableCoin and deposit these into the Votium contract based on the provided gauge list and corresponding allocations.

Fees are received from the PrimaryFeeAggregator via bridging from the primary chain to the secondary chain. The cross-chain flow is highlighted below within "Appendix: Cross-Chain Compose Message"

Appendix: Cross-Chain Compose Message

A general overview of the flow for LayerZero's compose message feature:



The specific flow for sending compose messages to successfully distribute fees is as follows:

> PrimaryFeeAggregator.processWeeklyDistribution
-> LocalReceiverBase.notifyFees

> LocalReceiverBase.notifyNewFees
-> LzComposeForwarder._notifyNewFees
-> OFT.send

> OFT.send
-> _debit
-> _debitView
-> _burn
-> _buildMsgAndOptions
-> encode
-> combineOptions
-> MsgInspector.inspect
-> _lzSend
-> _payNative
-> Endpoint.send

> Endpoint.send
-> _send
-> _outbound
-> getSendLibrary
-> SendLib.send
->_suppliedNative
-> _assertMessagingFee
-> _payNative

> Offchain

> Endpoint.verify
-> _inbound

> Endpoint.lzReceive

> _clearPayload
> OFT.lzReceive

> OFT.lzReceive
> _lzReceive
-> _credit
-> _mint
-> encode
-> Endpoint.sendCompose

> Endpoint.sendCompose

> Endpoint.lzCompose
> LzComposeReceiverBase.lzCompose

> LzComposeReceiverBase.lzCompose
> _notifyNewFees

> VotiumFeeReceiver._notifyNewFees
> Votium.depositUnevenSplitGaugesSimple

Appendix: Votium Usage

The ReceiverVotium contract deposits funds for specific gauges into the Votium contract via the depositUnevenSplitGaugesSimple function. The rationale behind this logic is to deposit tokens as voting incentives for different gauges.
Whenever a round has been surpassed, funds are transferred to a distributor contract where these are likely distributed to users based on their votes.

With this methodology, fees will be used to incentivize users to vote for gauges.

Privileged Functions

- transferToken
- setTokenApproval
- setGauges

Issue_23	Funds within Votium contract can remain permanently stuck if not processed within a round
Severity	**High**
Description	The Votium contract exposes a withdrawUnprocessed function which allows the initial depositor (ReceiverVotium contract in this case) to withdraw any funds which have not been processed in the corresponding round. This can happen if a gauge was killed which is presumably handled in a different contract but then rolled over to this contract whenever the endRound function is called with the _gauges array. However, the ReceiverVotium contract does not expose such a function call towards the Votium contract. In such a scenario, funds will remain in the Votium contract without the possibility to ever withdraw them. The same goes for the recycleUnprocessed function.
Recommendations	Consider implementing a withdrawUnprocessed call to the Votium contract to recover eventually stuck funds.
Comments / Resolution	Resolved.

Issue_24	Call of depositUnevenSplitGaugesSimple can revert due to change of allowToken settings within the Votium contract
Severity	**Low**
Description	Whenever the depositUnevenSplitGaugesSimple function is invoked on the Votium contract, the _takeDeposit function will be invoked which then transfers the token in. This function exposes the following check: *if (requireAllowlist == true) {* *require(tokenAllowed[_token] == true, "!allowlist");* *}* Furthermore, governance of the Votium contract can deny previously allowed tokens: *function allowToken(address _token, bool _allow) public onlyTeam {* *tokenAllowed[_token] = _allow;* *emit TokenAllow(_token, _allow);* *}* In such a scenario, the depositUnevenSplitGaugesSimple call will revert. Fortunately, that will not expose an issue due to the fact that the cross-chain-transfer has already been successfully finalized.
Recommendations	Since the only negative impact in that scenario is the need to manually recover stuck tokens, we do not recommend any change. However, it is optionally possible to check the current allowance setting beforehand and in such a case where allowance is denied for

	this token, simply transfer the token back to the fee recipient contract.
Comments / Resolution	Acknowledged.

Issue_25	Game Theory: Anyone can trigger incentive creation
Severity	**Low**
Description	As we have elaborated within “Appendix: Votium Usage”, the idea behind this methodology is to create incentives for votes. While this is out of scope, the possibility remains that a malicious user creates incentives right before the end of a round, then vote accordingly and pockets the majority of the just deposited incentives.
Recommendations	Consider if this can become a problem, if yes that would mean the fee procession must become permissionless.
Comments / Resolution	Acknowledged.

StableStaker

The StableStaker contract is a vault-akin contract which allows users to deposit the stableCoin in exchange for a receipt token which is also an OFT.

Contrary to the standard ERC4646 vault concept, the value aggregation of this contract is not periodically but rather streamlined over the course of 2 days whenever new fees are accrued. Similar to the Votium Module and VoteMarket module, fees are received from the PrimaryFeeAggregator. These fees are in the form of the stableCoin and will be used to increase the underlying value of the vault.

Users can deposit funds into the contract via the deposit and mint function and withdraw funds from the contract by either invoking cooldownAssets() or cooldownShares() with then subsequently invoking the unstake function once the cooldown period has been surpassed.

Appendix: Streamline Process:
As within most vaults, the exchange rate is determined via the underlying balance, which is mostly stored in the totalAssets() function. Exchange rate calculations are the following:

> shares = assets * supply / totalAssets()

> assets = shares * totalAssets() / supply

With most vaults, any fees/rewards are simply added to the return value of totalAssets(). In fact, most of the time this is simply the balance of the underlying ERC20 token. Such a practice is vulnerable for flash-theft attacks where malicious users deposit a large amount of tokens right before a reward distribution and withdraw immediately afterwards, profiting from a beneficial exchange rate after rewards have been distributed.

The StableStaker vault does not follow that practice, instead, rewards are streamlined linearly over the period of two days which prevents such flash-theft attacks.

Whenever the PrimaryFeeAggregator processes fees, the notifyNewFees amount within the StableStaker is invoked which will store how much rewards for the upcoming week are received (*including eventual leftover rewards from non-created streams):

uint256 weekAmount = asset.balanceOf(address(this)) - totalStoredAssets - totalCooldownAssets - residualAmount;

lastWeeklyAmountReceived = weekAmount;

A new reward stream over the next 2 days is then created by the pro-rata amount based on the days which have been passed since the beginning of the week and the lastWeeklyAmountReceived value:

*uint256 updateDays = getDay() - (getWeek() * 7) + 1;*

*uint256 newAmount = (weekAmount / 7) * updateDays;*

If we are for example in the 7th day of a week when the notifyNewFees function is invoked, a new reward stream with the full weekly rewards is created for the next 48 hours.
On another hand, on the 1st day of a week, a reward stream with 1/7th of the weekly rewards is created for the next 48 hours.

On each contract interaction, the _updateDailyStream function is invoked with the goal to ensure that there is always an active stream if lastWeeklyAmountReceived was not yet fully consumed. This is done by fetching the upper limit as last possible day for a week when rewards can be updated:

*uint256 lastUpdateDay = (lastDistributionDay / 7 + 1) * 7 - 1;*

The logic here is trivial, if the distribution is from the first week, lastUpdateDay will become 6, if the distribution is from the second week, lastUpdateDay will be 13 and so on and so forth. This basically acts as an upper safeguard to not accidentally distribute more than lastWeeklyAmountReceived within a single week.

If in any update, either within notifyNewFees or _updateDailyStream, there is a pending reward stream, the leftover tokens from this reward stream will be calculated and incorporated in the new stream.

*Edge-Case: Leftover rewards from last week:

Whenever lastWeeklyAmountReceived has not been completely consumed in the stream creation logic in the case where _updateDailyStream is not invoked until lastUpdateDay, any remaining undistributed funds will flow into the lastWeeklyAmountReceived variable for the upcoming fee distribution.

The following cases were identified and formally verified:

SCENARIO REWARD NOTIFICATIONS

SCENARIO 1: notifyNewFees is invoked the very first time

SCENARIO 2: notifyNewFees is invoked subsequent times with no remaining streams ad no remaining weekly rewards

SCENARIO 3: notifyNewFees is invoked subsequent times with remaining streams but consumed weekly rewards

SCENARIO 4: notifyNewFees is invoked subsequent times with expired streams and remaining weekly rewards

SCENARIO 5: notifyNewFees is invoked subsequent times with remaining streams and remaining weekly rewards

SCENARIO STREAM UPDATES

SCENARIO 1: _updateDailyStream is invoked before 24 hours have passed after last update

SCENARIO 2: _updateDailyStream is invoked after 24 hours have passed and before end of the week (within 7th day)

SCENARIO 3: _updateDailyStream is invoked after 24 hours have passed and somewhere in next week

Appendix: Cooldown Methodology:

Contrary to the standard vault contract, this contract exposes a cooldown period before users can actually withdraw their assets. Users can either invoke the cooldownAssets or cooldownShares functions to burn their shares and create a withdraw "request". This request can then be fulfilled after the cooldownDuration has surpassed by calling unstake().

During the request creation via the _cooldown function, all important storage adjustments such as decreasing totalStoredAssets and decreasing totalMintedSupply are already handled such that during the actual withdrawal only the cooldowns mapping for the user is deleted and funds are transferred out.

This means that any funds which are in the cooldown period will not further accumulate value from fees.

Privileged Functions

- setPeer
- setDelegate
- setPreCrime
- setEnforcedOptions
- setMsgInspector
- setPeers
- setDefaultOptions
- setBridgeEnabled
- setCooldownDuration
- setFeeAggregator
- setRewardRegulator
- setGovStaker

Issue_26	Governance Privilege: Setting of peer
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, the setPeer function can be used to add a malicious peer and then receive tokens from another chain without having burned the origin amount. (This would only work if original tokens have been bridged out from the main chain to side-chains by any user)
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_27	Vault Inflation Attack: Malicious user can manipulate the exchange rate for the deposit function
Severity	**Medium**
Description	The standard calculation within the deposit function to calculate the received amount of shares is as follows: *assets * totalMintedSupply / totalAssets()* While the totalAssets() function is not trivially manipulatable by donating the token, it is still theoretically possible to inflate the divisor due to the fee accrual process. This will then result in the standard vault inflation attack where it is possible to manipulate the ratio in such a way that it rounds significantly down for subsequent users (zero shares are forbidden). This can especially become an issue if fees are accrued while there has been no deposit made yet.
Recommendations	Consider implementing a minSharesOut parameter and/or mint an initial amount to the dead address.
Comments / Resolution	Acknowledged, the first deposit will be made by governance.

Issue_28	Edge-case where unused rewards from last week are rolled over to next week will result in decreased reward rate
Severity	**Low**
Description	There is one specific edge-case scenario where the lastWeeklyRewardsAmount was not completely used for stream updates within 7 days after notifyNewRewards was called. More specifically, this scenario can happen when the _updateDailyStream function was not invoked post beginning of 7th day after reward distribution (getDay = 6). This will then result in 1/7th of lastWeeklyRewardsAmount being rolled over to the next week. If we trivially consider lastWeeklyRewardsAmount being 70e18, that means 10e18 will be rolled over to the next week. During the normal business logic, if the _updateDailyStream function is invoked during the 7th day, it would streamline these 10e18 rewards over the next 2 days. If this is however not happening but instead the notifyNewFees amount is invoked during day 8 (getDay = 7), these 10e18 tokens are not streamlined over 2 days but are rather incorporated into weekAmount (assuming next week also receives 70e18 tokens): *uint256 weekAmount = asset.balanceOf(address(this)) - totalStoredAssets - totalCooldownAssets - residualAmount;* > 140e18 - 60e18 - 0 - 0 > 80e18 This means 80e18 tokens are now allocated to 11.42e18 tokens / day. The result of this is that these 10e18 tokens which would have

	previously been distributed over 2 days, will now be distributed over 7 days, significantly flattening the reward curve. A similar issue is present if during week 1 (as example) 100e18 tokens are present to distribute. On the 7th day all 100e18 tokens would then be distributed over the next 48 hours. If we however don't distribute them (at all) during week 1 and wait until week 2 and there was only an influx of 10e18 tokens, the next created reward stream (as example on the first day of this week) would be created with 110e18 / 7 * 1 tokens, which is significantly smaller than the reward stream which would have been created on the 7th day during week 1.
Recommendations	We consider this as a design choice and as expected by the developer. However, in our opinion it is still important to mention this to ensure it is desired and can be safely acknowledged.
Comments / Resolution	Acknowledged.

Issue_29	Decreased coolDownDuration can be used to exit earlier
Severity	**Low**
Description	Whenever the _cooldown function is invoked, the following struct will be set to a user's cooldown mapping: *cooldowns[msg.sender] = AssetCooldown({* *underlyingAmount:* *uint224(cooldowns[msg.sender].underlyingAmount + assets),* *cooldownEnd: cooldownEnd* *});* Notably, cooldownEnd is determined as follows: *uint32 cooldownEnd = uint32(block.timestamp + cooldownDuration);* So far, it is perfectly handled that the cooldownEnd is calculated using the cooldownDuration as this ensures even if the cooldownDuration is changed and a position has already been prepared for withdrawal, the actual withdrawal cannot happen earlier (due to the usage of cooldownDuration at the time of the withdrawal preparal). This can however be tricked by simply invoking the _cooldown function again because then it will override the old cooldownEnd using the potentially smaller cooldownDuration which results in the possibility to withdraw earlier. This can either be an intended design-choice or an unexpected side-effect. However, in our opinion the withdrawal time should always be determined by the cooldownDuration at the time of the withdrawal initiation, even if the cooldownDuration is decreased/changed at a later point.

Recommendations	Consider if this is expected by design or if this is unintended. In the latter scenario, the logic must be refactored to not allow for overriding a pending withdrawal.
Comments / Resolution	Acknowledged.

Issue_30	Lack of address(0) check for multiple setter functions
Severity	**Low**
Description	The contract exposes multiple setter functions, namely: setFeeAggregator setRewardRegulator setGovStaker None of these functions has an address(0) validation which can result in a DoS of the business logic.
Recommendations	Consider implementing such a validation.
Comments / Resolution	Acknowledged.

Issue_31	govStaker address is not set upon deployment
Severity	**Low**
Description	The _setNewStream function invokes the StakerRewardRegulator contract to fetch the amount of tokens which shall be allocated to the vault contract. The leftover amount is meant to be allocated to the GovStaker contract: *if (govAmount > 0) {* *address _govStaker = govStaker;* *asset.transfer(_govStaker, govAmount);* *IFeeReceiver(_govStaker).notifyNewFees(govAmount);* *}* However, during the deployment, the GovStaker contract is accidentally not set which means the contract does not work without governance calling setGovStaker() first.
Recommendations	Consider setting the govStaker address during the constructor.
Comments / Resolution	Acknowledged.

Issue_32	convertToShares() does not handle edge-case where shares are non-zero and totalAssets() returns zero
Severity	**Low**
Description	After careful consideration we were not able to reproduce such a state where shares are non-zero and totalAssets() returns zero, as this would effectively mean that circulating shares are unbacked. While this may happen with vaults which can decrease in underlying value, this vault does not expose such a scenario. Nevertheless, the original ERC4646 implementation accounts for such a scenario: *function _convertToShares(uint256 assets, Math.Rounding rounding) internal view virtual returns (uint256 shares) {* *uint256 supply = totalSupply();* *return* *(assets == 0 || supply == 0)* *? _initialConvertToShares(assets, rounding)* *: assets.mulDiv(supply, totalAssets(), rounding);* *}* In our opinion, this codebase should be as near as possible to the standard ERC4646 implementation and therefore the same edge-case should be considered. A similar issue is present within previewWithdraw(), however, that does not expose a problem because if that would be adjusted it means users will simply burn shares for zero output amount, therefore the current implementation is even beneficial for users as they are forced to wait until totalAssets() becomes != 0 to withdraw.
Recommendations	Consider following the same practice as highlighted above.

Comments / Resolution	Acknowledged.

Issue_33	Lack of RewardRegulatorSet event during constructor
Severity	**Informational**
Description	The RewardRegulatorSet event is not emitted during contract deployment. Important events should be emitted to allow users to retrace all flows.
Recommendations	Consider emitting the said event.
Comments / Resolution	Resolved.

StakerRewardRegulator

The StakerRewardRegulator contract is a simple helper contract which is responsible for calculating the share of fees which is allocated to the vault and the share which is allocated to the GovStaker contract. The contract owner can set all important boundaries.

Appendix: Staker Reward Calculation

Whenever a new reward stream is created within the StableStaker contract, the nominal amount for this reward stream is determined as:

> newAmount = (lastWeeklyAmount / 7) * updateDays

Based on this amount, a conversion is applied which is determined by the price of the stableCoin token. There are three different scenarios.

a) Price is above/equal maxPrice: use maxStakerPct to calculate the pro-rata amount:
> amount * maxStakerPct / MAX_PCT

b) Price is below/equal minPrice: use MinStakerPct to calculate the pro-rata amount:
> amount * minStakerPct / MAX_PCT

c) Price is below maxPrice and above minPrice: The following mathematical formula is used to derive the stakerPct:

> stakerPct = ((price - minPrice) * MAX_PCT) / (maxPrice - minPrice);
> Calculate difference between price and minPrice
> Multiply result by 10_000
> Divide result by price range

> stakerPct = minStakerPct + ((stakerPct * (maxStakerPct - minStakerPct)) / MAX_PCT);
> Calculate the difference between the maximum and minimum staker percentages
> Scales it by stakerPct from previous step
> Divides by MAX_PCT

> Add to minStakerPct

The core logic of this is to calculate stakerPct based on the price position of the current price in the range between minPrice and maxPrice.
This means the price position scaled by MAX_PCT is multiplied with the delta of (maxPrice - minPrice) and then applied on minStakerPct. The higher the price, the higher will be stakerPct and more funds are being allocated to the GovStaker.

EXAMPLE VALUES:
* We are using extreme values, ignoring MAX_PRICE_RANGE to highlight the logic

minPrice = 0.8e18
maxPrice = 1.1e18
minStakerPct = 7000
maxStakerPct = 9000
price = 0.9e18

The price is at 33.3% range between 0.8e18 and 1.1e18. This means we apply 33.3% on (maxStakerPct-minStakerPct) and add this to minStakerPct to represent the stakerPct value.

7000 + (0.333 * (9000-7000)) = 7666

Using the formula in the code:

stakerPct = ((price - minPrice) * MAX_PCT) / (maxPrice - minPrice);
stakerPct = ((0.9e18 - 0.8e18) * 10_000) / (1.1e18 - 0.8e18);
stakerPct = (0.1e18 * 10_000) / 0.3e18;
stakerPct = 3333

stakerPct = minStakerPct + ((stakerPct * (maxStakerPct - minStakerPct)) / MAX_PCT);
stakerPct = 7000 + ((3333 * (9000 - 7000)) / 10_000);
stakerPct = 7000 + (6666000 / 10_000);
stakerPct = 7666

Privileged Functions

- setPriceBounds
- setStakerPctBounds

No issues found.

Base/Dependencies

CoreOwnable

The CoreOwnable contract serves as a handler contract which aggregates all important data from the ProtocolCore contract:

(https://github.com/defidotmoney/dfm-contracts/blob/58e83d59544f57af2719295e1d025a6b032b6c08/contracts/base/ProtocolCore.sol)

Notably, it exposes the following parameters:

- owner: The owner of the ProtocolCore contract
- bridgeRelay: The determined bridge relay
- feeReceiver: The determined fee recipient
- guardian: The determine guardian

Furthermore, it exposes necessary modifiers to ensure proper access controls:

- onlyOwner
- onlyBridgeRelay
- ownerOrGuardianToggle

In the current scope, the feeReceiver and guardian addresses as well as the onlyBridgeRelay and ownerOrGuardianToggle modifiers remain unused. However, we assume these variables are used in other contracts throughout the architecture that are not scope of this audit. This usage should be carefully checked and eventually audited as well.

Privileged Functions

- none

No issues found

SystemStart

The SystemStart contract is a simple auxiliary contract that fetches the START_TIME from the core protocol upon deployment and then exposes getter functions to return the amount of weeks and days passed since the START_TIME.

The getWeek and getDay functions are primarily used for the following purposes:

- Check for bridge frequency
- Check if fees have been already distributed for a week
- Used for reward calculation within the StableStaker contract

Appendix: Truncation

In an effort to receive the current day / week, the time which has passed is simply divided by the corresponding divisor (day/week). This will then trivially return the amount of days/weeks which have passed since the start time. The truncation always happens based on the START_TIME.

Privileged Functions

- none

No issues found.

Bridge/Dependencies

BridgeTokenBase

The BridgeTokenBase contract extends LayerZero's OFT standard:

https://github.com/LayerZero-Labs/LayerZero-v2/tree/main/packages/layerzero-v2/evm

The following additions/modifications have been introduced:

- Bridging was made pausable

- A sendSimple function was introduced which allows for bridging tokens without the option of a compose message, using native ETH as the fee token. A corresponding quoteSimple function was developed.

- Access Control was revised: Only the owner within the CORE_OWNER contract can execute onlyOwner functions. The functions transferOwnership and renounceOwnership were overridden with a revert.

- An enumerableSet for all EIDs has been implemented

- Default options for msgType = 1 (SEND) can be set. These are specifically used within the sendSimple function.

This audit partially includes the OFT contract but does not involve an in-depth audit of it.

Appendix: Default Options

As explained, the contract exposes a _setDefaultOptions function which allows the contract owner to set the defaultOptions variable:

```
function _setDefaultOptions(bytes memory options) internal {
if (options.length > 0) _assertOptionsType3(options);
defaultOptions = options;
}
```

The defaultOptions variable is then used whenever a new peer is added as enforcedOption for the corresponding EID and msgType = 1 which translates into SEND (simple cross-chain transfers without a compose message):

```
enforcedOptions[_eid][1] = defaultOptions;
```

This ensures that the correct amount of gas is quoted and not accidentally more (as it would be the case with compose messages).

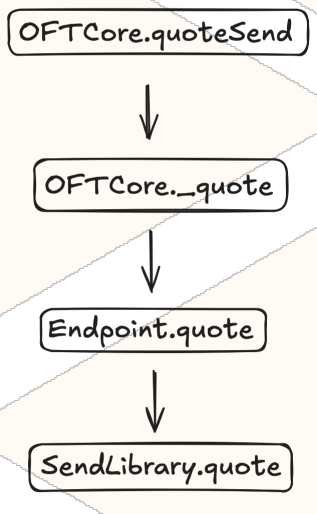
Appendix: Quoting Mechanism

Cross-Chain messages and transfers require executors to execute actions on the destination chain. Since these executions cost gas, there is a need to fund these transactions.

In LayerZero, the funding of transactions is directly handled whenever a cross-chain transaction is initiated, either with provided msg.value to the Endpoint.send call or via a transferFrom of the LzToken from the msg.sender to the Endpoint.

In an effort to provide an accurate quotation of how much a cross-chain transaction will cost, the LayerZero Endpoint provides a quote function which gauges the gas cost based on the MessagingParams.

The default flow to gauge gas cost is as follows:



Privileged Functions

- setPeer
- setDelegate
- setPreCrime
- setEnforcedOptions
- setMsgInspector
- setPeers
- setDefaultOptions
- transferOwnership
- renounceOwnership

Issue_34	Governance Privilege: Setting of peer
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, the setPeer function can be used to add a malicious peer and then receive tokens from another chain without having burned the origin amount.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_35	Access Control inconsistency between setPeer and setPeers
Severity	**Low**
Description	The setPeer and setPeers functions are logically the same, the only difference is that the latter function allows for setting multiple iterations in the same function call. The first function uses the following access control mechanism: *require(msg.sender == owner() || msg.sender == CORE_OWNER.bridgeRelay(), "DFM:T Only owner or relay")* while the second function uses the onlyOwner modifier. Therefore, the first access control mechanism is loose, allowing the relayer to change peers.
Recommendations	Consider using the onlyOwner modifier for setPeer() as well.
Comments / Resolution	Acknowledged.

Issue_36	sendSimple() does not return OFTReceipt
Severity	**Informational**
Description	The sendSimple function allows users to trivially invoke cross-chain transfers using the native token as fund token. Contrary to the send function, it only returns amountSentLD instead of the OFTReceipt struct.
Recommendations	Consider following L0's practice and returning the OFTReceipt.
Comments / Resolution	Acknowledged.

Issue_37	Bridge process will not work on chains where alternative endpoint is deployed
Severity	**Informational**
Description	LayerZero has an alternative endpoint contract for blockchains where the native token is worthless. In such a scenario the _payNative function is overridden to transfer in an alternative token: https://github.com/LayerZero-Labs/LayerZero-v2/blob/main/packages/layerzero-v2/evm/protocol/contracts/EndpointV2Alt.sol#L10 While this does not apply to this scope, it should still be considered when deploying on other blockchains.
Recommendations	Consider keeping this scenario in mind.
Comments / Resolution	Acknowledged.

Issue_38	Old OFT version is used for current implementation
Severity	**Informational**
Description	Currently, the following L0 versions are used: "@layerzerolabs/lz-evm-oapp-v2": "=2.3.16", "@layerzerolabs/lz-evm-protocol-v2": "=2.3.16", These are outdated and do not align with the current version: https://github.com/LayerZero-Labs/LayerZero-v2/tree/592625b9e5967643853476445ffe0e777360b906/packages/layerzero-v2/evm Most specifically, this can be seen here: The sendSimple function invokes the _debit function as follows: *_debit(msg.sender, _amount, 0, _eid);* which then invokes the override _debit function: *function _debit(* *address _from,* *uint256 _amountLD,* *uint256 _minAmountLD,* *uint32 _dstEid* *) internal virtual override returns (uint256 amountSentLD, uint256 amountReceivedLD) {* *require(isBridgeEnabled, "DFM:T Bridging disabled");* *return super._debit(_from, _amountLD, _minAmountLD, _dstEid);* *}* The most recent implementation of the OFT _debit function has the following function selector:

	function _debit(*uint256 _amountLD,* *uint256 _minAmountLD,* *uint32 _dstEid* *)* While there are - from our knowledge - no issues with the previous version, we always recommend using the most updated version. (exceptions prove the rule)
Recommendations	Given that the whole testing suite was based on the current implemented version, we do not recommend a change at this point. However, for the future it might make sense to always incorporate the most recent version.
Comments / Resolution	Acknowledged.