BAIL security

Parallel Protocol V3 Core

FINAL REPORT

March ‘2025

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	Parallel Protocol V3
Website	parallel.best
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/parallel-protocol/parallel-core/tree/2491020713a7a634b32fea1a6fd1eadafbb300b1 https://github.com/parallel-protocol/parallel-parallelizer/tree/c56bbe62d8ce8a421a6851d0e63bae0b58f32a44 https://github.com/parallel-protocol/parrallel-tokens/tree/4473277ff40e8cb21aeda727c3126fa648f1f1e4
Resolution 1	https://github.com/parallel-protocol/parallel-parallelizer/tree/f1fef64ecb68ad61b94e3e80dd7e2493e583379c https://github.com/parallel-protocol/parrallel-tokens/tree/0198dec89345c2f1d713cfb7f535cf6df1fd284a

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	2	2			
Medium	12	4		8	
Low	19	10		9	
Informational	18	10		8	
Governance	8	4		4	
Total	59	30		29	

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 pose 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

parallel-tokens/TokenP.sol

The TokenP.sol contract is the actual stablecoin contract used in the system. This is an upgradable ERC20 contract that implements permit functionality and adds access control functionality via a separate _accessManager contract. It also adds mint and burn functions, allowing restricted contracts to mint and burn tokens to users. The parallelizer module mints and burns tokens in exchange for collateral.

Privileged Functions

- burnSelf
- burnFrom
- mint

No issues found.

parallel-tokens/BridgeableTokenP.sol

The BridgeableTokenP.sol contract is a Layerzero OFT token, which extends the functionality of the TokenP contract by giving it multi-chain capabilities. Users can bridge the stablecoin between chains using this contract and are minted placeholder OFT tokens in case the bridge limits have been met on any chain. The contract also implements a swapLzTokenToPrincipalToken function, allowing users to swap in these placeholder OFT tokens for the actual stablecoins later once the chain has limits to spare.

The contract tracks a creditDebitBalance, which measures the difference between inflows and outflows of a particular chain, as well as daily and global mint and debit limits. These values determine if the end-user gets credited stablecoins or OFT tokens. So once the credit/debit limits are hit, instead of stopping the bridging process, users can keep bridging and collecting placeholder OFT tokens without disturbing the chain mint/burn limits. Once there is room in the chain, users can swap in their OFT tokens for the stablecoin via the swapLzTokenToPrincipalToken function.

Core Invariants:
INV 1: The tokenP mint burn limits must respect globalDebitLimit, dailyDebitLimit, globalCreditLimit, and dailyCreditLimit

Privileged Functions

- emergencyRescue
- toggleIsolateMode
- setFeesRate
- setDailyCreditLimit
- setGlobalCreditLimit
- setDailyDebitLimit
- setGlobalDebitLimit
- setFeesRecipient
- pause
- unpause

Issue_01	globalCreditLimit can be bypassed
Severity	**High**
Description	In the _calculatePrincipalTokenAmountToCredit function, the passed-in _amount value is cast into int256. *principalTokenAmountToCredit = int256(_amount) + creditDebitBalance > int256(globalCreditLimit)* *? uint256(int256(globalCreditLimit) - creditDebitBalance)* *: _amount;* If the _amount passed in is larger than int256.max, this leads to an overflow and int256(_amount) evaluates to a negative number. In this case, instead of adding the amount to creditDebitBalance, it is subtracted from it, so even if the mint violates the globalCreditLimit, this check cannot stop it. Assume currently the creditDebitBalance is 99e18, and globalCreditLimit is 100e18. So a maximum of 1e18 tokens can be minted on the chain.

	However, if a user calls swapLzTokenToPrincipalToken with the _amount set to uint(256).max, it casts it into an int256 resulting in -1. The code quoted above evaluates to _amount, and the principal token credited evaluates to the dailyCreditLimit, assuming the dailyUsage of that day was 0. *if (dailyUsage + principalTokenAmountToCredit > dailyCreditLimit) {* *principalTokenAmountToCredit = dailyCreditLimit > dailyUsage ? dailyCreditLimit - dailyUsage : 0;* *}* So the user will be able to mint dailyCreditLimit amount of tokens instead of the expected 1e18 tokens. If dailyCreditLimit is set to 10e18, they will be able to mint 10e18 tokens, bypassing the global credit limit and pushing the creditDebitBalance value beyond it.
Recommendations	Use the Openzeppelin SafeCast library to prevent under/overflows during casting.
Comments / Resolution	Resolved by following recommendations.

Issue_02	In BridgeableTokenP, LZ messages can be received (causing tokens to be credited) even when the contract is paused.
Severity	**Medium**
Description	The _lzReceive() function does not have the `whenNotPaused` modifier, allowing LZ messages to be received (causing tokens to be credited) even when the contract is paused.
Recommendations	If unintended, consider adding the `whenNotPaused` modifier to the _lzReceive() function to ensure that this cannot occur. If intentional (e.g. to prevent cross-chain disruptions), this issue can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_03	Fee rate changes can lead to unexpected number of received tokens
Severity	**Low**
Description	When swapping and bridging, users are charged a fee based on the feesRate. However, there is no guarantee that when the user submits his transaction, the fee won't be changed by the owner. Even though the owner is trusted, this can still happen accidentally, for example: 1. User submits a swap for 100 tokens and there is 10% fee 2. Owner decides to change the fee to 20% at the same time 3. Owner transaction executes first and the user receives less than expected 4. There is also no slippage and the user couldn't have done much to protect against this

Recommendations	Consider implement slippage during swapping. During bridging, this can still happen so performing fee rate changes should be declared beforehand and not changed drastically.
Comments / Resolution	Acknowledged.

Issue_04	_handleCreditPrincipalToken has an incorrect natspec
Severity	**Informational**
Description	The natspec for _handleCreditPrincipalToken has the _to input and the _amountLD input, however the _isFeeApplicable param is not there.
Recommendations	Add _isFeeApplicable to the natspec
Comments / Resolution	Resolved by following Recommendations.

Issue_05	Outdated comment in _creditPrincipalToken
Severity	**Informational**
Description	The comment above _creditPrincipalToken states that if the contract does not have enough balance, it will mint the token amount to the receiver. However, this is outdated functionality and the comment needs to be changed. The contract now only mints tokens to the receiver.
Recommendations	Change the comment accordingly or change the functionality to match the comment.
Comments / Resolution	Resolved by following Recommendations.

Issue_06	Incorrect comment in _debit()
Severity	**Informational**
Description	In _debit(), the comment following comment is wrong: /// @dev Assert that the final creditDebitBalance is greater than the globalDebitLimit. if (creditDebitBalance < globalDebitLimit) revert ErrorsLib.GlobalDebitLimitReached(); Based on the code it references, the comment should say “greater than or equal to”
Recommendations	Consider adjusting the comment to match the code it references
Comments / Resolution	Resolved by following Recommendations.

Issue_07	if feesRate=0, bridge can be done repeatedly to DOS swaps from LZ->Principal token
Severity	**Informational**
Description	If feesRate == 0, a griefer can repeatedly bridge between chains A<->B so that dailyCreditAmount[_getCurrentDay()] and dailyDebitLimit[_getCurrentDay()] are met, pausing the bridging process for everyone else. The only cost here for the griefer would be the gas and LZ fees, since the feesRate is 0.
Recommendations	Consider seting feesRate to be non-zero, or acknowledging this vector.
Comments / Resolution	Acknowledged.

Issue_08	BridgeableTokenP does not support TokenP's permit functionality
Severity	Informational
Description	TokenP inherits ERC20Permit but BridgeableTokenP does not support using permit to get the approvals. So there will always be two transactions involved in order to bridge.
Recommendations	Consider adding a sendWithPermit function which users the permit function on TokenP to gain approvals
Comments / Resolution	Resolved by following recommendations.

parallel-tokens/FlashParallelToken.sol

The FlashParallelToken contract allows users to flash loan stablecoin tokens for a fee. This contract also has mint access to the TokenP contract and can mint and burn them at will. When the flashLoan function is called, the requested number of tokens are minted to the receiver address and then a callback function is called on the msg.sender. After the callback, the initial amount + fee amount of tokens is taken out of the receiver address and burnt.

The contract itself follows the ERC3156FlashLender standard and expects the caller to follow the ERC3156FlashBorrower standard.

Core Invariants:

INV 1: Contract balance after flashloan call must be higher than the balance before (for non-zero fees).

Privileged Functions

- setFlashLoanParameters
- setFlashLoanFeeRecipient
- toggleActiveToken

Issue_09	accrueInterestToFeeRecipient attempts to transfer an incorrect amount of tokens
Severity	**High**
Description	The accrueInterestToFeeRecipient function can be called by anyone to collect the accrued fees in the Flash loan contract and send it forward to the recipient's address. The flashloan contract supports multiple tokens and has a balance variable to sum up the fees from the various tokens. *function accrueInterestToFeeRecipient(address[] calldata tokens) external returns (uint256 balance) {* *for (uint256 i = 0; i < tokens.length; i++) {* *IERC20 token = IERC20(tokens[i]);* *balance += token.balanceOf(address(this));* *token.safeTransfer(flashLoanFeeRecipient, balance);* *}* The issue is that balance sums up the values from different tokens, and then tries to send off the balance amount to the recipient. If the flashloan contract supports multiple tokens, then this transfer call will fail, since when transferring the second token, it will try to send the balance of the first token as well. Say the contract has 10 USDC and 10 USDT as fees. In the first iteration, balance will be 10 and send forward 10 USDC. In the second iteration, balance accumulates to 20 and will try to send forward 20 USDT tokens, which don't exist leading to a revert.
Recommendations	Consider setting the balance for each iteration, instead of incrementing it.
Comments / Resolution	Resolved by following recommendations.

Issue_10	flashFee function does not follow ERC3156 specification
Severity	**Low**
Description	The ERC 3156 specification states the following: *"If the token is not supported, flashFee MUST revert.* *If a token is not supported then the flashFee function must revert"* This is not followed in the implementation here. *function _flashFee(address token, uint256 amount) internal view returns (uint256) {* *return amount.percentMul(tokenMap[token].feesRate);* *}* The function does not check if the passed token is active or not. If unconfigured, it returns 0 instead of reverting, and if configured but marked inactive, it returns whatever the stored rate is. This violates the ERC 3156 pattern.
Recommendations	Consider reverting if the token is not supported to conform to the specification.
Comments / Resolution	Resolved by following recommendations.

Issue_11	maxFlashLoan does not follow ERC 3156 specification
Severity	Low
Description	The ERC 3156 specification states the following: *"If a token is not currently supported maxFlashLoan MUST return 0, instead of reverting."* In the current implementation, the maxFlashLoan function returns 0 for unconfigured token addresses. However, if tokens have been configured but then set to inactive via toggleActiveToken, this will return the configured value instead of returning 0.
Recommendations	Consider setting maxBorrowable to 0 when inactivating a token, or have the maxFlashLoan check for the state of the token. This issue can also be acknowledged since flashloans cannot be taken out during a flash loan.
Comments / Resolution	Resolved by following recommendations.

parallel-parallelizer/BaseHarvester.sol

The harvester contracts are used to rebalance the composition of the Parallelizer system. The BaseHarvester contract implements common configuration and calculation functions inherited by the actual harvester implementations. When a collateral exceeds its target exposure, it is converted to its yield-bearing form or vice-versa, and the amount converted is calculated by the _computeRebalanceAmount function. This enables the system to optimize yield revenue.

Privileged Functions

- setYieldBearingAssetData
- setMaxSlippage
- toggleTrusted
- recoverERC20
- setTargetExposure

Issue_12	_computeRebalanceAmount doesn't completely rebalance the system
Severity	**Low**
Description	The _computeRebalanceAmount function calculates the disbalance in the system, and calculates the amount of token that need to be converted from one collateral backing to another for the system to be bealanced. It does this by calculating the difference between the current exposure of the collateral and the target exposure. The issue is that while it calculates a stablecoin amount that needs to be converted, it does not take into consideration the mint/burn fees. Due to this, the amount returned by this function will always fall short of completely balancing the system.
Recommendations	Consider acknowledging that completely balancing the system can involve a lot of computation on-chain. The adjustYieldExposure function can be directly used to rebalance the system by calculating the amounts off-chain.
Comments / Resolution	Acknowledged.

parallel-parallelizer/GenericHarvester.sol

The GenericHarvester contract implements a generic rebalancer contract to rebalance the Parallelizer system. It inherits the BaseHarvester contract, inheriting the ability to calculate the amount of disbalance in the system. The harvest function computes the imbalance through the _computeRebalanceAmount function and then calls adjustYieldExposure. The adjustYieldExposure function takes a flashloan of stablecoins, burns them for assetA through the Parallelizer system, swaps assetA for assetB via a swap router, and then mints back the stablecoins using assetB to pay off the flashloan. This allows the system to move its exposure from assetA to assetB in a single transaction.

Privileged Functions

- setTokenTransferAddress
- setSwapRouter

Issue_13	harvest Uses Stale Exposure Limits Due to Missing Synchronization with Parallelizer
Severity	**Medium**
Description	The BaseHarvester contract supports two types of exposure configurations for yield-bearing assets: static, hardcoded values, or dynamic values retrieved from the Parallelizer contract. When an asset is configured with overrideExposures = 2, the harvester is expected to fetch **live exposure limits** from Parallelizer before performing any rebalancing actions. However, the harvest function does **not call** updateLimitExposuresYieldAsset, the function responsible for syncing the current exposure limits from Parallelizer. As a result, the harvester may continue to use **stale exposure parameters**, even after governance has updated them on the Parallelizer. This discrepancy creates a risk where rebalancing operations are based on **outdated exposure boundaries**, violating the updated constraints set by governance. In practice, this could lead to either overexposure or underexposure of some assets. Because the update is not automatic, the correct values will only be used **if someone manually calls** updateLimitExposuresYieldAsset, which is inconsistent with the intended dynamic behavior.
Recommendations	To ensure rebalancing decisions are made using accurate governance-defined parameters, it is recommended to **automatically call** updateLimitExposuresYieldAsset for each yield-bearing asset at the start of the harvest function whenever overrideExposures = 2. This ensures the harvester always operates with up-to-date exposure limits and aligns with governance intent.

Comments / Resolution	Resolved by following recommendations.

Issue_14	Rebalance Interference Between Harvester Contracts
Severity	**Medium**
Description	The protocol allows concurrent rebalancing operations through both MultiBlockHarvester and GenericHarvester. Since MultiBlockHarvester's rebalancing occurs across multiple blocks (with a delay between harvest and finalizeRebalance), users can call GenericHarvester's harvest during this period, using inaccurate exposure data and causing incorrect final allocations. MultiBlockHarvester's rebalancing process happens in two steps: 1. harvest() initiates a rebalance, swapping assets and potentially requesting redemptions 2. finalizeRebalance() completes the process after external actions complete During the period between these steps, the Parallelizer's exposure data is in a transient state, but GenericHarvester's harvest function operates based on this incomplete exposure data. Consider the following simplified scenario: 1. Parallelizer has the assets USDC and USDM a. The total stablecoins minted are 100 b. USDM has minted 50 tokenP (50%) c. USDC has minted 20 tokenP (20%) 2. On the harvester contracts, the target exposure for USDM is 60%.

	3. The trusted role on MultiBlockHarvester calls harvest and starts a rebalance by swapping 10 tokenP to USDC. a. The total stablecoins minted are now 90 b. USDM has minted 50 tokenP (55.5%) c. USDC has minted 10 tokenP (11.1%) 4. While the multi-block rebalance hasn't finished, a user calls harvest on GenericHarvester. a. Given that USDM exposure is 55.5% and should be 60%, the user swaps 4 USDC to USDM b. Now, USDM has minted 54 tokenP (60%) c. USDC has minted 6 tokenP (6.6%) 5. After a while, the multi-block rebalance ends and the trusted role calls finalizeRebalance on MultiBlockHarvester, which swaps 10 USDM to tokenP: a. The total stablecoins minted are now 100 again. b. USDM has minted 64 tokenP (64%) c. USDC has minted 6 tokenP (6%) 6. After the rebalances are finished, the USDM exposure has surpassed the target exposure because a rebalance occurred while a multi-block rebalance was ongoing. When users call GenericHarvester's harvest during an ongoing MultiBlockHarvester rebalance: - The calculation of needed rebalance amounts will be based on intermediate (inaccurate) exposure data - Multiple rebalances may target the same exposure goal, resulting in overshooting after all operations complete - Final token allocations will deviate from governance's intended target exposures

Recommendations	It's recommended to not allow users to call harvest on both contracts while there is a multi-block harvest ongoing.
Comments / Resolution	Acknowledged.

Issue_15	Unprotected adjustYieldExposure Bypasses Exposure Controls and Slippage Protection
Severity	**Low**
Description	The protocol's rebalancing flow is designed as follows: The harvest function calculates proper rebalance amounts based on governance-set target exposures. However, the adjustYieldExposure function has public visibility with no access controls. This allows any user to call adjustYieldExposure directly with arbitrary parameters, circumventing: - Target exposure calculations based on governance settings - Slippage protections derived from the maxSlippage parameter
Recommendations	If adjustYieldExposure is left as an external function, users are expected to do the rebalancing calculations and slippage calculations themselves or by the UI. Otherwise, this function can be made internal and only accessible via the harvest function.
Comments / Resolution	Resolved by following recommendations.

Issue_16	Harvesting will be DoSed while there are flash loan fees on tokenP
Severity	**Low**
Description	Harvesting on GenericHarvester will only work when tokenP has no flash fees at all. However, the flash fees are applied to everyone without exception, which will cause a DoS on GenericHarvester when the governance wants to collect some flash fees.
Recommendations	it's recommended to allow exempting some addresses from paying flash fees, such as GenericHarvester.
Comments / Resolution	Acknowledged.

Issue_17	Guardian Can Steal Funds by Updating swapRouter and tokenTransferAddress
Severity	**Governance**
Description	According to the protocol's documentation, the **guardian role is not trusted to extract funds** and any malicious behavior should be recoverable by the governor. However, the current implementation violates this assumption by allowing the guardian to manipulate two critical configuration variables in the GenericHarvester contract: swapRouter and tokenTransferAddress. Both variables are protected by a restricted modifier, which—per the original Angle codebase—grants access to the guardian. This creates a serious vulnerability, as the guardian can use these functions to route funds out of the contract and bypass the intended security model. Exploit path: 1. The guardian calls setSwapRouter, maliciously setting it to an ERC20 token contract (e.g., the USDC token).

	2. The guardian then calls adjustYieldExposure, supplying **crafted calldata** designed to make the contract approve its entire USDC balance to the guardian's address. 3. During execution, _performRouterSwap blindly forwards this calldata to the swapRouter address—now pointing to the USDC token contract. 4. As a result, the contract calls approve(guardian, amount) on USDC. 5. The guardian then simply calls transferFrom to move the tokens into their own wallet. This effectively enables the **guardian to drain all ERC20 tokens** from the GenericHarvester contract in a single transaction, fully **circumventing governance oversight**. A similar attack is possible by abusing setTokenTransferAddress, which can also be redirected to malicious or unintended contracts for approval calls. This vulnerability is particularly dangerous because the GenericHarvester is explicitly designed to **hold protocol and user funds**—often as part of yield rebalancing operations. The attack can be repeated across different tokens, giving a malicious guardian the ability to steal the entire balance of the contract.
Recommendations	Consider restricting setSwapRouter and setTokenTransferAddress so that **only the governor** has permission to update these variables.
Comments / Resolution	Resolved by following recommendations.

parallel-parallelizer/MultiBlockHarvester.sol

The MultiBlockHarvester is another rebalancer contract implementation that is completely access-controlled. This contract is used to rebalance between collateral tokens which cannot be immediately converted to one another at a good rate. For example, RWA based tokens generally implement a KYC-enabled redemption method where stablecoins like USDC can be obtained at no slippage but takes time. The contract implements a 2-step rebalance method, where the first harvest call simply burns stablecoins for the collateral which can then be swapped into other tokens. Then the rebalance is completed by calling the finalizeRebalance function where the new collateral tokens are used to mint back the stablecoins from the Parallelizer.

Privileged Functions

- setYieldBearingToDepositAddress
- finalizeRebalance
- harvest

Issue_18	DoS Risk During Harvest When a Stablecoin Depegs
Severity	Medium
Description	The MultiBlockHarvester contract uses the _checkSlippage function during harvest operations to verify that swap results remain within an acceptable slippage range. This check compares the input and output amounts of a swap, normalized to the same number of decimals, using a maxSlippage threshold. However, the current logic assumes that all stablecoins involved are **fully pegged to each other**, which may not always hold true. In practice, stablecoins can **temporarily depeg** from their target value, as seen in multiple past incidents. This assumption can lead to incorrect behavior: **- Swapping a depegged stablecoin for a pegged one:** The slippage check may incorrectly fail, even if the actual USD value exchanged is accurate—causing the transaction to revert. **- Swapping a pegged stablecoin for a depegged one:** The slippage check may incorrectly pass, even if the value received is significantly lower than expected. As a result, harvest operations can fail (DoS) or proceed with poor value received, depending on the direction of the swap.
Recommendations	To ensure robust slippage checks, convert both the input and output amounts to their **USD-equivalent values** using up-to-date price oracles before comparing them in _checkSlippage. Alternatively, this can be acknowledge if it is expected that no depeg happens.
Comments / Resolution	Acknowledged.

Issue_19	Harvesting USDM May Revert Due to Receiving 1 Wei Less Than Expected
Severity	**Medium**
Description	USDM exhibits the same behavior as stETH in that it may transfer **1 wei less than expected**, due to rounding errors in its internal logic. (link to code). This discrepancy causes a problem during the harvest process. When reducing exposure to USDM, the contract swaps tokenP for USDM, expecting to receive an exact amountOut. It then attempts to forward that full amount to the deposit address. However, because the USDM transfer delivers **1 wei less** than amountOut, the contract ends up trying to transfer **more tokens than it actually holds**, causing the transaction to revert and resulting in a denial of service (DoS) for that action. This issue will also happen on rebalances within GenericHarvester.
Recommendations	When handling USDM, avoid relying on the amountOut value. Instead, transfer the **entire USDM balance held by the contract**, which ensures compatibility with the token's rounding behavior and prevents reverts during harvest.
Comments / Resolution	Resolved by following recommendations.

Issue_20	Unnecessary allowance in _rebalance function
Severity	Low
Description	The _rebalance function grants allowance of tokenP tokens to the parallelizer contract. This allowance is unnecessary, since the parallelizer contract uses the burnSelf function to burn tokenP tokens from the caller, which does not need allowance.
Recommendations	Consider removing the tokenP allowance granting to the Parallelizer contract.
Comments / Resolution	Resolved by following recommendations.

Issue_21	Having the same maxSlippage value for all assets is not efficient
Severity	Low
Description	The function _checkSlippage always uses the same slippage threshold for all tokens. However, each operation uses different tokens with different exchange methods, which may incur on different slippage value depending on the tokens exchanged. For this reason, it's recommended that maxSlippage would rather be a mapping than a single uint variable.
Recommendations	Consider acknowledging this issue or add logic to support different maxSlippage based on the asset.
Comments / Resolution	Resolved by following recommendations.

Issue_22	Hardcoded addresses can break functionality on other chains
Severity	**Informational**
Description	The contracts have hardcoded address corresponding to mainnet. However if the system is deployed on some other chain, the address would need to be changed or would lead to incorrect results. *address constant USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;*
Recommendations	When deploying on chains other than mainnet, change the hardcoded addresses.
Comments / Resolution	Acknowledged.

Issue_23	Guardian can steal all funds by updating yieldBearingToDepositAddress
Severity	**Governance**
Description	According to the README, the guardian should not be allowed to take out funds from the system. However, the guardian is allowed to call setYieldBearingToDepositAddress and steal all funds from the contract. The vulnerability stems from the permission structure in the MultiBlockHarvester contract. According to the README, guardians should not be able to take out funds from the system. However, the contract design gives guardians excessive privileges through two key functions: 1. The setYieldBearingToDepositAddress function allows guardians to change where funds are deposited during rebalancing operations 2. The toggleTrusted function lets guardians grant trusted status to any address, including their own. A malicious guardian can execute a three-step attack to drain funds: 1. Grant themselves trusted status using the guardian-accessible function 2. Update the deposit address to point to an address they control 3. Initiate a harvest operation, causing funds to be sent to their controlled address This attack completely bypasses intended protocol safeguards and contradicts the security model described in the documentation.
Recommendations	Consider updating the setYieldBearingToDepositAddress function to be governance-only rather than guardian-accessible. This ensures that only protocol governance (not guardians) can modify

	deposit addresses, aligning with the intended security model and preventing unauthorized fund extraction.
Comments / Resolution	Resolved by following recommendations.

parallel-parallelizer/DiamondProxy.sol

The DiamondProxy contract implements EIP2535 Diamond proxy. This proxy pattern allows contracts to have the functionality of multiple contracts at the same time, allowing for a way to bypass the 24kb contract size limit in a way. Here, the Diamond contract stores a mapping of function selectors and contract addresses, and when a user calls a particular function, the proxy contract delegates that call to the stored address.

The base DiamondProxy contract simply has a fallback function that fetches a contract address from storage based on the passed msg.sig and delegates the call to the fetched address.

Issue_24	Parallelizer does not support delayed actions
Severity	**Medium**
Description	The Parallelizer system uses the Openzeppelin's AccessManager contract for access control. This contract allows delayed operations, where users can queue an operation and only execute it after a certain amount of time has passed. The contract tries to support this as is evident in the checkCanCall function. *(bool immediate, uint32 delay) =* *AuthorityUtils.canCallWithDelay(address(accessManager), caller, address(this), bytes4(data[0:4]));* *if (!immediate) {* *if (delay > 0) {* *ParallelizerStorage storage ts = s.transmuterStorage();* *ts.consumingSchedule = true;* *accessManager.consumeScheduledOp(caller, data);* *ts.consumingSchedule = false;* *}* The issue is that for delayed operations, the consumeScheduledOp function is called on the AccessManager contract which in turn calls isConsumingScheduledOp on the target contract, which in this case is the Parallellizer system. However, this callback function is not present in the Parallelizer system, making delayed actions impossible.
Recommendations	Consider implementing the isConsumingScheduledOp function in the Diamond proxy to support delayed operations.
Comments / Resolution	Resolved by following recommendations.

Issue_25	Incorrect TRANSMUTER_STORAGE_POSITION hash on Constants
Severity	**Informational**
Description	The storage on Parallelizer is namespaced, meaning it's stored in a specific location, determined by the following hash: > keccak256("diamond.standard.parallelizer.storage") - 1 >0xc1f2f38dde3351ac0a64934139e816326caa800303a1235dc53707d0de05d8bd However, the resulting hash is not the same as the one used on Parallelizer. If we compute the hash stated in the comments in Chisel, the result is the following: >0x4b2dd303f68b99d244b702089c802b6e9ea1b5d4ef61fd436d6c41abb1178c75
Recommendations	Even though it doesn't have any impact, it's recommended to correct where the Parallelizer storage is stored.
Comments / Resolution	Resolved by following recommendations.

Issue_26	The system doesn't support fee-on-transfer tokens
Severity	**Informational**
Description	Most functions in the system expect entire token amounts to be transferred in transferFrom calls. This is however not the case if the tokens used implement a transfer-tax. In such a case, the system will have incorrect accounting leading to a loss.
Recommendations	Do not support any tokens implementing transfer-tax as collateral.
Comments / Resolution	Acknowledged.

parallel-parallelizer/Diamond proxy facets

The DiamondCut, DiamondEtherscan, and DiamondLoupe contracts along with the libraries LibDiamond, and LibDiamondEtherscan are standard Diamond proxy pattern functions following EIP2535. These were forked from the original diamond proxy implementation by Mudgen.

The LibDiamond library implements the checkCanCall function, which does access control checks using the Openzeppelin AccessManager contract. This library also implements the diamondCut function, allowing admins to add/remove facets to the proxy. Admins pass in an array of type FacetCut which contains the address of the implementation (facet) and the function selectors that will be delegated to that implementation.

The DiamondEtherscan contract and the LibDiamondEtherscan library are used to generate a dummy contract with function selectors from all the facets and store it following EIP1967. This enables Etherscan, which knows about EIP1967, to display all functions in the diamond proxy on the website and allows users to interact with the contract directly.

The DiamondLoupe contract is full of view functions displaying the various facets and function selectors supported by the diamond proxy.

Privileged Functions

- diamondCut
- setDummyImplementation

No issues found.

parallel-parallelizer/AccessManagedModifiers.sol

The AccessManagedModifiers contract implements 2 access control methods. The first is the restricted modifier, which uses the checkCanCall function to check if the caller is allowed to call the function in question. This utilizes an external Openzeppelin AccessManaged contract to handle access control levels.

The contract also implements the nonReentrant function. This prevents re-entrancy in methods decorated with this modifier by storing the entered state in the contract storage.

No issues found.

parallel-parallelizer/Getters

The Getters contract implements various view-only functions to communicate data stored in the proxy contract. Since the data is stored in specific storage slots following the upgradable pattern, it can be difficult to read the values stored in these slots. This contract creates simple getter functions to access information about the various collateral configurations, oracle data, and trusted roles.

No issues found.

parallel-parallelizer/Redeemer.sol

The Redeemer contract is used to redeem stablecoins for the underlying collateral tokens. When redeeming, users send in their stablecoins which are burnt and the redeemer contract sends back the constituent collateral tokens. The amounts of the tokens sent back depend on the proportion of the collateral assets present in the system.

The system also implements a penalty factor to prevent bank runs during transitory de-collateralization events. Users receive less collateral than the current collateral ratio would indicate, increasing the collateral ratio of the system with subsequent redemptions. This deters users from exiting the system during transitory downturns due to the penalty and incentivizes the remaining users by increasing the collateralization ratio of the system, giving

a better deal to later exits.

Core Invariants:
INV 1: The collateral ratio of the system must always increase or stay the same with redemptions (depending on the settings of the piecewise linear function).

Privileged Functions
- updateNormalizer

Issue_27	Users might be forced into forfeiting a collateral
Severity	**Medium**
Description	Other than regular collateral tokens, there are also managed collaterals which work slightly differently and are being interacted with using the LibManager library. Each such collateral can have more than 1 subcollateral. For example, collateral A might have subcollaterals B, C and D. If one of those subcollaterals is another regular collateral, then a user might be forced into forfeiting his collaterals without actually having to. Imagine the following scenario: 1. There are 3 collaterals - A, B and C, collateral A is managed and requires a whitelist, and has token B as a subcollateral, the collaterals B and C do not require a whitelist 2. Bob wants to redeem and the loop will iterate over [A, B, B, C] as B is both a subcollateral and a collateral 3. Bob is not whitelisted for A so he uses the forfeiting functionality made specifically for redeeming when a user is not whitelisted for one of the collaterals, he specifies A and its subcollateral B as the function checks token A's whitelist for each of its subcollaterals (as indexCollateral only changes when all collateral's subcollaterals are passed)

	4. This causes Bob to forfeit the collateral B as well, resulting in a loss of funds for him
Recommendations	Consider not having any collateral which can be both a raw collateral as well as a subcollateral of some other managed collateral.
Comments / Resolution	Acknowledged.

parallel-parallelizer/RewardHandler.sol

The RewardHandler contract is used to sell off any accrued tokens by the contract for other tokens, preferably collateral tokens to increase backing. This is implemented in the sellRewards function, where the One Inch router is called to carry out a swap transaction. The caller must be allowed by the system or set as a trusted seller, and the governance must give a prior allowance of that token to the One Inch router contract for the transaction to succeed. The function does not allow the sale of any collateral tokens, and does pre and post-balance checks to ensure the same.

Core Invariants:

INV 1: The amount of collateral tokens in the contract must not decrease after a One-inch router call.

Privileged Functions

- sellRewards

Issue_28	Guardian Can Self-Sandwich sellRewards to Steal Most of the Rewards
Severity	**Governance**
Description	The sellRewards function allows a *guardian* or trusted role to swap accrued rewards on the Parallelizer contract into collateral tokens. To prevent poor execution, the function includes: - A minAmountOut parameter to protect against excessive slippage. - A check to ensure that at least one supported collateral token balance increases post-swap. However, since the **guardian is not a fully trusted role**, these protections are insufficient. The guardian can execute a **self-sandwich attack** to drain most of the rewards while still passing the post-swap checks. **Attack Overview:** 1. The guardian selects a DEX pool and **manipulates its price** (e.g., by trading into it to make it imbalanced) 2. They call sellRewards, executing the swap through the manipulated pool. 3. Due to the manipulated pricing, the contract receives only a **small fraction** of the rewards' actual value. 4. The post-swap check passes as long as the balance of at least one collateral token has increased (even by a few wei). 5. The guardian then **rebalances the pool** and captures the arbitrage profit—effectively **stealing most of the rewards**. The attack can be further optimized by using a **custom 1inch executor** that skips the actual swap entirely. Instead, it sends a

	minimal amount of the expected collateral directly to the contract, avoiding swap fees and leaving the majority of the rewards untouched. This vulnerability allows the guardian to **extract most of the rewards from** Parallelizer, undermining the protocol's ability to accumulate value from its yield strategies.
Recommendations	The sellRewards function should **not be callable by an untrusted role** like the guardian. Consider limiting this functionality to only the governance.
Comments / Resolution	Acknowledged.

Issue_29	Funds can be left uninvested
Severity	**Governance**
Description	The sellRewards function allows an admin to sell non-collateral tokens in the contract for other tokens, preferably collateral tokens to increase the backing. This swap is done via the ONE_INCH_ROUTER.call call. The issue is that if the funds are simply swapped in the Parallelizer contract, this might not count towards the collateral backing. This is because some collaterals can be configured to be managed, in which case the tokens aren't kept in the contract itself but in an external asset manager contract. This is reflected in the LibManager library, which has the invest function used to send funds to the asset manager during swaps. Thus if a collateral is managed, the collateral tokens obtained from the result of the swap need to be sent to the manager contract of that collateral, and not to the Parallelizer contract itself.
Recommendations	The OneInch router payload should be set such that the resultant tokens go to the asset manager contract in case of externally managed collateral.
Comments / Resolution	Resolved.

Issue_30	Function sellRewards Vulnerable to Reentrancy, Allowing Collateral Theft.
Severity	**Governance**
Description	The sellRewards function is intended to allow a trusted role or the *guardian* to sell accrued rewards via 1inch and swap them into collateral tokens, without putting the protocol's funds at risk. To prevent abuse, the function checks that the balance of collateral tokens in the Parallelizer contract does **not decrease** during the swap. This is done by comparing the collateral balances before and after the call, ensuring no tokens have left the contract during the process. However, this protection can be **bypassed via a reentrancy attack**, allowing a malicious guardian to **steal collateral** by manipulating the internal balance during execution. Attack overview: 1. The guardian calls sellRewards, providing a custom **executor** address—this is a feature supported by 1inch that allows arbitrary logic to be executed during the swap. 2. Within the swap execution, the guardian's custom executor reenters the Parallelizer and calls swapExactInput, converting some collateral to tokenP. 3. This increases the collateral balance mid-execution. 4. Since the balance check in sellRewards is performed after the swap, the increase makes it appear as though no tokens were stolen. 5. The guardian can then transfer the increased collateral amount to their own address, **bypassing the balance delta check**.

	This exploit assumes that at least one collateral token has an existing **approval** to the 1inch router, which is a realistic scenario in practice.
Recommendations	Consider adding the nonReentrant modifier to the sellRewards function to prevent reentrancy
Comments / Resolution	Resolved by following recommendations.

Issue_31	Function sellRewards Will Revert on Sonic Due to Hardcoded 1inch Router Address
Severity	**Medium**
Description	The sellRewards function allows a guardian or trusted address to swap accrued rewards on Parallelizer into collateral tokens using the 1inch router. However, the router address is **hardcoded** in the contract, limiting its functionality to chains where 1inch has deployed the router at that exact address. According to the Parallel development team, the target deployment chains include: **Ethereum Mainnet, Optimism, Arbitrum, Polygon PoS, Base, and Sonic**. While 1inch uses the same router address on many chains, **it is not currently deployed on Sonic**. As a result, the sellRewards function will revert on Sonic due to the missing contract at the hardcoded address—leading to **stuck rewards** and a denial of service (DoS) for reward-selling operations.
Recommendations	Make the 1inch router address configurable via a **governance setter** or constructor parameter. This will ensure compatibility across all target chains, including those where 1inch is not yet deployed or uses a different address.

Comments / Resolution	Acknowledged.

parallel-parallelizer/Setters

The SettersGovernor and SettersGuardian contracts allow admins to set various parameters of the system. These functions allow adding/removing collateral, changing fee and penalty parameters, a token recovery function recoverERC20, and pause toggles.

Privileged Functions

- togglePause
- setFees
- setRedemptionCurveParams
- toggleWhitelist
- setStablecoinCap
- recoverERC20
- setAccessManager
- setCollateralManager
- changeAllowance
- toggleTrusted
- addCollateral
- adjustStablecoins
- revokeCollateral
- setOracle
- setWhitelistStatus

Issue_32	Zero Fees Enable Exposure Manipulation Across Collaterals
Severity	**Medium**
Description	In Parallelizer, mint and burn fees are dynamically adjusted based on collateral exposure levels. However, when **zero fees** are configured for both **minting and burning** within the same exposure range on a specific collateral, while other collaterals have non-zero fees, it creates an opportunity for **exposure manipulation**. Example scenario: - The protocol supports two collaterals: **USDC** and **USDM**. **-USDC:** -Current exposure: 60% **-Mint fee:** 0% until 80% exposure **-Burn fee:** 0% **-USDM:** -Current exposure: 40% **-Mint fee:** 0% until 30% exposure **-Mint fee:** 100% above 30% exposure **Note:** The above configuration resembles a live configuration on Angle's Transmuter, from which Parallelizer is forked. This highlights that such misconfigurations are not just theoretical but have precedent in production deployments. At this point, no one can mint USDM without incurring a 100% mint fee, as its exposure exceeds 30%. However, an attacker can bypass this restriction by manipulating the exposure levels: Attack flow: 1. The attacker **mints USDC** until its exposure reaches 80%, paying **zero fees**.

	2. This **dilutes USDM's exposure** from 40% down to 20%. 3. Now, the attacker **mints USDM** up to 30% exposure again, paying **zero fees**. 4. Finally, the attacker **burns the USDC** previously minted, again paying **zero fees**. As a result, the attacker increases USDM's exposure **above its original level** without incurring **any fees**—effectively **bypassing the fee gate** designed to discourage overexposure. This exploit undermines the protocol's fee-based exposure control mechanism and allows attackers to **manipulate exposure ratios** to mint favored assets at no cost.
Recommendations	To prevent this form of manipulation: - Avoid overlapping **zero-fee segments** for both **mint and burn** on any collateral. - Ensure that mint and burn fee schedules are designed so that **any round-trip manipulation incurs a non-zero cost**, even during low-exposure segments. - Consider applying a **minimum baseline fee** or dynamic adjustment based on recent exposure volatility.
Comments / Resolution	Acknowledged.

Issue_33	Setting a Collateral Manager with Existing Funds Causes a Temporary Collateral Ratio Drop
Severity	**Medium**
Description	In Parallelizer, governance has the ability to assign an external manager to handle a specific collateral token. Once this manager is set, the system no longer accounts for that token using the balance held directly in Parallelizer. Instead, it relies on the manager's totalAssets function to report the amount of collateral under management. However, when a new manager is set, the existing funds in Parallelizer are **not automatically transferred** to the manager contract. As a result, although the collateral is now classified as managed, the manager itself still holds zero tokens. This creates a temporary state where the system's view of available collateral is significantly understated. The totalAssets function will return zero, even though there are actual funds still held in Parallelizer. During this period, the protocol will compute a lower-than-accurate **Collateral Ratio (CR)**. If a user initiates a redemption while the system is in this state, the calculated CR will underestimate the actual backing, and the user will receive less value than they should. This discrepancy leads to a tangible loss for users, even though the funds are present in the system.
Recommendations	To prevent this situation, it is recommended that the full balance of the collateral token be automatically transferred from Parallelizer to the newly assigned manager at the moment the manager is set. This ensures that the accounting via totalAssets is immediately accurate and that redemptions and other collateral-sensitive operations reflect the true state of the system.
Comments / Resolution	Acknowledged.

Issue_34	Updating or removing a collateral manager can be griefed
Severity	**Low**
Description	When updating or removing a collateral manager, it first checks that totalAssets on that manager is strictly zero. This can be griefed as anyone could donate directly to the collateral manager contract so that totalAssets would not be zero. It also can be griefed by frontrunning the admin call and swapping to deposit some collateral in that collateral manager, which would cause totalAssets to increase. This same issue is also present on revokeCollateral, where it also checks that totalAssets is zero if there is a manager.
Recommendations	It's recommended to pass a flag argument on setCollateralManager to indicate if the previous balance on the external manager should be ignored.
Comments / Resolution	Resolved by following recommendations.

Issue_35	Revoking a collateral can be griefed
Severity	**Low**
Description	When revoking a collateral, it first checks that the total stablecoins minted with that collateral must be strictly zero. However, this can be griefed by frontrunning the governance call and swapping a tiny amount of collateral for tokenP, causing an increase in the stables minted with that collateral, and preventing its revocation.
Recommendations	It's recommended to pass a flag argument on revokeCollateral to indicate whether to ignore that check.
Comments / Resolution	Resolved by following recommendations.

Issue_36	setWhitelistStatus does not clear previous whitelist data
Severity	**Informational**
Description	Collaterals can have a whitelist which can be removed/added using setWhitelistStatus which has the following logic: *if (whitelistStatus == 1) {* *// Sanity check* *LibWhitelist.checkWhitelist(whitelistData, address(this));* *collatInfo.whitelistData = whitelistData;* *}* *collatInfo.onlyWhitelisted = whitelistStatus;* When adding a whitelist, the whitelist data is set to the new one. However, in the scenario where a whitelist item is being removed, the whitelist data stays. Any integrators/users calling getCollateralWhitelistData will still receive the outdated whitelist data which can cause issues for them and confuse them in regards to whether there is an active whitelist or not.

Recommendations	Clear the whitelist data upon removing the whitelist item
Comments / Resolution	Resolved by following recommendations.

Issue_37	Incorrect revert error when revoking a collateral
Severity	**Informational**
Description	When revoking a collateral, we have the following check: *if (collatInfo.decimals == 0 || collatInfo.normalizedStables > 0) revert NotCollateral();* The revert reason is that the collateral revoked is not actually a collateral. Reverting due to left side condition will indeed occur when that is not a collateral, however the reason for reverting due to the right side is not when the address provided is not a collateral as having stablecoins minted from it indicates exactly the opposite.
Recommendations	Introduce a new custom error and split the checks into 2, reverting with the appropriate error.
Comments / Resolution	Resolved by following recommendations.

Issue_38	No code length check in setAccessManager
Severity	**Informational**
Description	The setAccessManager function in the LibSetters library is used to set the access manager contract. since the Openzeppelin AccessManager contract is being used, the passed-in address is expected to be a contract. However, the setter function never checks if the passed-in address is a contract or just an EOA. This is in contrast to how the access manager is set in the AccessManaged contract, which comes with the AccessManager contract. Here, the setAuthority function makes sure that the code length in the passed address is non-zero before setting it.
Recommendations	Consider adding a similar code-length check in the setAccessManager function.
Comments / Resolution	Resolved by following recommendations.

Issue_39	Admin can drain the system
Severity	**Governance**
Description	The recoverERC20 function can be used to take out any collateral from the system. This can be carried out by a malicious admin or a bad actor who gained access to the private keys of the admin account. Since this can entirely brick the system, special care should be taken in determining the access to this function. This function should only be callable by a time-locked multisig, to minimize the threat to the system.
Recommendations	Consider setting up a time-locked multisig to do important governance actions in the protocol. This helps immensely in case of a private-key leak.

Comments / Resolution	Acknowledged.

Issue_40	Zero or Negative Fees Enable Arbitrage via Oracle Deviations
Severity	**Governance**
Description	The Parallelizer protocol allows configurable swap fees, including the possibility of setting **zero or even negative fees** under certain conditions. However, this flexibility introduces a critical vulnerability: **arbitrage via oracle price deviations**. - The **guardian** role has the ability to set **zero fees**. Since this role is **not fully trusted**, a malicious guardian could set zero fees and **exploit oracle mispricings** to arbitrage the system and extract value from the reserves. - When **negative fees** are set by governance, the system includes a basic safeguard to ensure that the negative value is not large enough to be directly profitable. However, this check: **- Does not account for deviations between oracle-reported prices and actual market prices**. - As a result, an attacker can still profit from small mispricings, **even when the negative fee check passes**. In both scenarios, the attacker can repeatedly arbitrage on Parallelizer by executing cost swaps using oracle prices that slightly differ from the real market prices due to its allowed deviations, profiting at the expense of the protocol.
Recommendations	To prevent this class of exploit, implement the following mitigations: **1. Guardian Fee Limitations** - Disallow the guardian from setting **zero fees**. - Introduce a **minimum fee buffer** that accounts for the **maximum**

	expected oracle deviation in the system. **2. Stronger Negative Fee Checks** - Enhance the negative fee validation logic to **include oracle deviation tolerance**. - Ensure that negative fees, even when individually safe, **cannot become profitable in combination with typical oracle inaccuracies**. By enforcing a conservative fee buffer that exceeds the highest oracle deviation observed across all price sources, the protocol can maintain protection against opportunistic arbitrage even in edge-case market conditions.
Comments / Resolution	Acknowledged.

parallel-parallelizer/Swapper.sol

The Swapper is the core of the Parallelizer system, where collateral tokens can be converted into stablecoins. Users can mint stablecoins by sending in collateral tokens, or receive collateral tokens by burning stablecoins. They are charged a fee for the transaction depending on the amount of exposure to that collateral.

The _quoteFees function calculates the mint and burn fees for a particular collateral depending on the system's exposure to that collateral. Similar to redemptions, a piecewise linear function is used to calculate the fee amount, interpolating based on the current exposure ratio. The _checkHardCaps function ensures no collateral exceeds its respective limit and the internal _swap function carries out the token transfers. In case the collateral is managed, this function also calls LibManager's invest function to send away the collateral to generate yield.

Core Invariants:
INV 1: Collateral amounts must be below their respective hard cap.

Issue_41	DoS on Mints When Using USDM or stETH as Managed Collateral
Severity	**Medium**
Description	In Parallelizer, certain collateral tokens can be marked as **managed**, meaning their balances are delegated to an external manager contract to generate yield. When a user mints tokens via the swapper and the collateral is managed, the received funds are automatically **forwarded to the manager** and passed into its invest function. Some tokens, such as **USDM** and **stETH**, exhibit non-standard behavior where they **transfer 1–2 wei less than the specified amount** due to internal rounding logic. This discrepancy becomes critical in the managed collateral flow: 1. The minting process sends the intended amount of collateral to the external manager. 2. The invest function is then called with the same amount that was "sent." 3. However, due to the token's behavior, the manager receives **slightly fewer tokens than expected**. 4. If the manager's implementation assumes it received the full amount and attempts to invest that exact value, the operation will **revert**. This results in a **DoS** for all mints involving USDM or stETH when they are marked as managed collateral. Since both tokens are used in other parts of the Parallelizer system, it's likely they will be used as collateral here as well.
Recommendations	Consider using wrapped versions of rebasing tokens, like wstETH to prevent rebase related errors.
Comments / Resolution	Acknowledged.

Issue_42	A deadline of 0 does not work as intended when using the permit functionality
Severity	**Low**
Description	According to the comment in Swapper, the 4 functions accepting a deadline must interpret a 0 deadline as no deadline. This is enforced by this code: *if (deadline != 0 && block.timestamp > deadline) revert TooLate();* Functions such as swapExactInputWithPermit (one of the 4 functions accepting a deadline) are also supposed to work like that and they enforce the rule the same way. However, the issue is that calling Permit2::permitTransferFrom during the flow uses the same provided deadline. As Permit2 does not have the same rule, this will simply revert as 0 is much less than the current block timestamp.
Recommendations	Using deadline as 0 to which implies no deadline is not compatible with the way Permit2 interprets deadlines. Consider always making sure non-zero values are used when creating permit2 payloads.
Comments / Resolution	Resolved by following recommendations.

Issue_43	First minter can put the protocol in an extreme exposure towards one collateral
Severity	**Low**

Description	When _quoteFees is called, we have the following code:
	
	if (normalizedStablesMem == 0 || n == 1) {
	return _computeFee(quoteType, amountStable, v.isMint ? collatInfo.yFeeMint[0] : collatInfo.yFeeBurn[0]);
	}
	
	If there are no stablecoins minted or if the fees are constant, then we simply apply it based on the first fee in the array. However, this allows the first depositor to put the protocol in an extreme exposure towards one collateral which can be hard to fix:
	
	1. Let's imagine that there are 3 collaterals and the ideal exposures for each are 33%
	
	2. The fees are crafted in such a way so that going over that charges you a big fee, disincentivizing users from minting with a collateral over that exposure rate
	
	3. The issue is that the first depositor is always charged flat based on the first fee, which means he could deposit a significant amount of funds from one collateral, let's imagine 1,000,000$ and he would not be charged any extra fee/penalty for minting such a huge amount
	
	4. For the exposure rate to go to the 33%, users must deposit 1,000,000$ of both of the other collaterals which will take a lot of time

Recommendations	Consider conducting the first deposit yourself (in a batch with the deployment), that way another potential vector (overflowing the collateral ratio which is a uint64 by having too many collateral relative to the minted stablecoins) is also handled.

Comments / Resolution	Acknowledged.

Issue_44	Missing whitelisted check for quoting functions
Severity	**Low**
Description	The quote functions within Swapper are designed to simulate a token swap and return the amounts that will result from that swap. The comments on quoteIn and quoteOut state that they should reflect the exact amounts if the swap is executed just after the quoting. However, these functions fail to check if a collateral token is whitelisted, so they don't revert when the caller is not whitelisted to trade with a gated token. After a non-whitelisted user calls quoteIn or quoteOut, the function should revert but it won't due to these missing checks.
Recommendations	Consider implementing whitelisted checks on quoteIn and quoteOut.
Comments / Resolution	Acknowledged.

Issue_45	Quote functions don't revert when trying to burn more stablecoins than allowed
Severity	**Low**
Description	The quote functions within Swapper are designed to simulate a token swap and return the amounts that will result from that swap. The comments on quoteIn and quoteOut state that they should reflect the exact amounts if the swap is executed just after the quoting. However, these functions fail to check whether the amount of stablecoins to burn is higher than the total amount of stables minted with that collateral. When this happens, quoteIn and quoteOut won't revert when they should.
Recommendations	Consider Implementing checks on quoteIn and quoteOut that ensure the burned stables are lower than the amount of stables minted with that collateral.
Comments / Resolution	Acknowledged.

Issue_46	Math formula comment in _quoteFees is incorrect
Severity	**Informational**
Description	In _quoteFees, we have the following formula as a comment: *m_t = [-1-g(0)+sqrt[(1+g(0))**2+2M(f_{i+1}-g(0))/b_{i+1}]]/((f_{i+1}-g(0))/b_{i+1})* While a bit hard to spot, the `]]` is actually incorrect and should be `])` instead to properly close the brackets.
Recommendations	Consider using]) instead.

Comments / Resolution	Resolved by following recommendations.

Issue_47	Incorrect attribution comment in Swapper facet
Severity	**Informational**
Description	The Swapper facet includes the following friendly fork attribution: */// @dev This contract is an authorized fork of Angle's `Swapper` contract* */// https://github.com/AngleProtocol/angle-transmuter/blob/main/contracts/parallelizer/facets/Swapper.sol* The link is invalid, and should be replaced with https://github.com/AngleProtocol/angle-transmuter/blob/main/contracts/transmuter/facets/Swapper.sol
Recommendations	Consider replacing the link with the correct one
Comments / Resolution	Resolved by following recommendations.

parallel-parallelizer/LibHelpers.sol

The LibHelpers library contains some helper functions to help with the mathematics in the system. The convertDecimalTo function is used to convert the decimal base of the amounts and the checkList function returns the index of an element from a list. The findLowerBound and piecewiseLinear functions are used to linearly interpolate between the set parameters when calculating the penalty factor during redemptions.

No issues found.

parallel-parallelizer/LibManager.sol

The LibManager library implements functions to interact with the manager contracts for collaterals. The Parallelizer system can either hold the collateral tokens itself or allow other contracts to hold the tokens on its behalf and perhaps invest them to earn more yield. The flow of control is decided for each collateral based on its isManaged setting.

When isManaged is set to 1 (or any non-zero number), the Parallelizer system invokes the functions in the LibManager library to handle the tokens. The invest and release functions are used to send tokens to the manager to earn yield or make them available to the Parallelizer system for withdrawals. The totalAssets function calculates the available collateral assets in the system which are being handled by the manager and used when calculating the collateral ratio of the system.

No issues found.

parallel-parallelizer/LibOracle.sol

The LibOracle handles the reading prices of the tokens from various Oracle sources. Some tokens can get their prices from other contracts and can be configured with external sources, in which case the readRedemption, readMint, or readBurn functions are called on other external contracts. Otherwise, this library itself handles the pricing. The read function handles the pricing from various supported sources such as:

1. Chainlink price feeds
2. PYTH price feeds
3. Morpho oracle
4. WSTETH pricing
5. CBETH pricing
6. RETH pricing
7. SFRXETH pricing

Items 1,2 and 3 from the list above are general pricing methods for various tokens. The rest are token-specific methods, reading the contract state for the relevant tokens. The contract also supports some other configured return values, such as OracleReadType.STABLE which always returns 1e18 and OracleReadType.MAX which always returns a stored value from the contract.

Issue_48	Missing Confidence Interval Check in PYTH Price Feeds
Severity	**Medium**
Description	The LibOracle library fetches token prices from multiple sources, including **PYTH**, by calling getPriceNoOlderThan. It performs some basic validations and normalization before returning the final price. However, the current implementation **does not validate the confidence interval** returned alongside the price. According to PYTH documentation, the confidence interval represents the estimated range within which the true market price lies, accounting for discrepancies across trading venues. When the confidence interval is **large relative to the price**, it indicates **high price uncertainty or volatility**. In such cases, using a single price value without accounting for the confidence range can result in **mispriced trades**. This opens the door for **MEV and arbitrage opportunities**: - During volatile periods, MEV bots can exploit the inaccurate pricing returned by LibOracle to **extract funds from the Parallelizer contract**. - The protocol will operate under the assumption that the PYTH price is accurate, even when the true market price may significantly differ.
Recommendations	Consider implementing a threshold for the confidence interval. If the confidence interval exceeds a reasonable threshold, it would indicate a volatile market and higher errors in pricing. Transactions could even be made to revert in this case if such scenarios are deemed too risky for the operation of the contracts.
Comments / Resolution	Resolved, no longer using pyth.

Issue_49	Attacker Can Arbitrage Using Two PYTH Prices Within the Same Block
Severity	Low
Description	PYTH is a **pull-based oracle**, meaning that its price feeds must be **explicitly updated** on-chain before the latest data can be read. A known behavior—documented in the PYTH docs—is that **two different PYTH prices can be used within the same block** if the feed is updated mid-transaction. This behavior introduces a **profit opportunity for attackers**, allowing them to manipulate pricing within a single block and extract funds from Parallelizer. Attack overview: 1. The attacker observes that a PYTH price has **not been updated recently** and diverges from the actual market price. 2. The attacker **executes a trade** on Parallelizer using the outdated PYTH price. 3. They then **update the PYTH feed** mid-block via the updatePriceFeeds endpoint. 4. Using the updated price, they perform a **reverse trade** on Parallelizer. This exploit is viable when: - The **price difference exceeds the protocol fees**, and - The original PYTH price is still **within the accepted freshness window** (i.e., not considered stale by governance thresholds). The attacker profits from the price discrepancy, minus fees.

	By executing this exploit, an attacker can arbitrage on Parallelizer and extract funds using two PYTH prices in the same block.
Recommendations	To mitigate this risk, consider the following defenses: **1. Use Keepers or Bots:** Deploy automated keepers to **proactively update** all relevant PYTH feeds before allowing price-dependent operations. This ensures that all users operate on the same updated price. **2. Enforce a One-Price-Per-Block Rule:** Cache the PYTH price at the start of a transaction or block and disallow in-transaction updates to ensure price consistency throughout a call. **3. Raise Protocol Fees:** Increase swap fees to make this type of arbitrage **economically unviable**.
Comments / Resolution	Resolved, no longer using pyth.

Issue_50	Missing Checks for Sequencer Uptime on Rollups
Severity	**Low**
Description	Many optimistic rollups—such as **Arbitrum** and **Optimism**—rely on **centralized sequencers** to handle transaction ordering, execution, and batching before submitting data to Ethereum (L1). When the sequencer is down, standard L2 interactions become unavailable to most users. While advanced users can still interact with the rollup through L1 contracts, this is expensive and complex, creating a **temporary monopoly** over L2 execution. This imbalance enables attackers to exploit **stale price data** and extract funds from Parallelizer.

	In this context, if Parallelizer continues operating during a sequencer downtime, **MEV bots or sophisticated actors** could exploit **outdated oracle prices** (e.g., from Chainlink or PYTH), and arbitrage mispriced tokens to extract funds from the protocol. This creates a significant **financial risk** to the protocol. The Parallelizer developers have confirmed deployment plans on **Optimism** and **Arbitrum**, both of which are affected by this sequencer availability issue.
Recommendations	Before querying oracle prices or executing sensitive operations on L2 chains: **- Check the sequencer uptime status**, using official uptime feeds (e.g., Chainlink's L2 Sequencer Uptime Feed). **- Pause or revert** any operations when the sequencer is down to prevent stale price exploitation.
Comments / Resolution	Acknowledged.

Issue_51	Circuit prices can have high errors
Severity	**Low**
Description	The read function allows chainlink data feeds and supports circuits of chainlink price feeds. It then goes through the multiple passed-in data feeds and multiplies them together to arrive at the final price. Basically, given chainlink feeds of A/B and B/C, it can find the price of A/C by multiplying them together. A similar approach is also used for the Pyth oracle setup. The issue is that this method adds together the deviations in each price feed. The ETH-USD chainlink price feed has a deviation threshold of 0.5%. If 3 such price feeds are used to calculate some price, the result will have a deviation of 1.5%. This can lead to bad swaps/redemptions.
Recommendations	Price feed circuits should be kept short, resulting in large errors otherwise.
Comments / Resolution	Acknowledged.

Issue_52	Missing min/max answer checks on Chainlink feeds
Severity	**Low**
Description	Some Chainlink feeds have built-in mechanisms that prevent the prices from updating when they arrive at their lower or upper limit. When a Chainlink feed asset arrives at a price that equals minPrice or maxPrice, it will stop reporting it. However, when querying the price from the endpoint, it will return the last updated price, which won't be the real market price. When this happens, attackers can use the stale prices from Chainlink to arbitrage on Parallelizer and extract funds.

Recommendations	Consider adding min/max checks for chainlink prices, and reverting when prices go out of those bounds.
Comments / Resolution	Acknowledged.

Issue_53	PYTH prices that return a positive expo can indicate errors
Severity	**Low**
Description	In all current PYTH feeds, the expo value is always negative because it represents the decimals that must be taken into account when interpreting the price. However, LibOracle also uses the PYTH prices even when the expo field is positive, and does different operations. Since no deployed PYTH oracle actually returns positive exponent, this can indicate errors with the oracle contract.
Recommendations	Consider acknowledging this, since although PYTH oracles can have positive exponents, there are none in practice.
Comments / Resolution	Resolved, no longer using pyth.

Issue_54	Presence of unnecessary pricing methods
Severity	**Informational**
Description	The LibOracle library implements pricing methods for STETH, CBETH, RETH, and SFRXETH. If all of these are not expected to be added as collateral, then some of these can be removed from the code.
Recommendations	The LibOracle library can be trimmed down by removing price feeds for collateral which isn't expected to be used.
Comments / Resolution	Acknowledged.

Issue_55	Oracles with hardcoded addresses will only work on mainnet
Severity	**Informational**
Description	OracleLib uses different hardcoded addresses to query on-chain prices. However, this hardcoded addresses will only work on mainnet and not in the rest of chains that Parallelizer will be deployed to (e.g. Op, arb, polygon, etc)
Recommendations	Consider acknowledging the issue and ensure the hardcoded values are correct for the respective deployment or allow these values to be changed and not be hardcoded.
Comments / Resolution	Acknowledged.

parallel-parallelizer/LibWhitelist.sol

The checkWhitelist library implements the checkWhitelist for access control. Certain collateral configurations can have its onlyWhitelisted flag set to 1. In this case the operator is subjected to another whitelist check done via this library method. This function simply extracts the stored keyringGuard address, and checks if the sender address is authorized.

No issues found.

parallel-parallelizer/Savings.sol

The BaseSavings and Savings contracts implement a fixed rate savings program utilizing the stablecoin. The contract extends an ERC4626 implementation for accounting deposits and withdrawals. An admin sets the interest rate of the contract by calling the setRate function. The contract then periodically mints itself stablecoins in its _accrue function, depending on the set interest rate and the time since the last accrual. This contract also requires mint access to the stablecoin contract.

Users are minted shares in 18 decimals to represent their deposits. To prevent inflation attacks, the contract mints itself some shares during creation to eliminate rounding errors.

Core Invariants:

INV 1: User funds should never decrease. Since the contract lacks a burn mechanism, the funds in terms of the stablecoin should never decrease outside of rounding errors of the order of asset decimals.

Privileged Functions

- togglePause
- toggleTrusted
- setRate
- setMaxRate

Issue_56	Burnt initial deposit accrues unrecoverable interest
Severity	**Informational**
Description	The Savings contract does a _deposit on its own behalf during contract creation. This is done to eliminate inflation attack vectors. The issue is that this initial deposit also accrues interest over time, which is unrecoverable. This leads to some permanently unrecoverable funds in the system.
Recommendations	If the fund amounts stuck are deemed insignificant, this can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_57	Function decimals will return the wrong value when the asset has less than 18 decimals
Severity	**Informational**
Description	The Savings token will always have 18 decimals as the initial deposit always mints 1e18 in exchange for 1 unit of the asset deposited. When that asset has less than 18 decimals, the Vault token will still have 18 decimals, but it's decimals function will just return the asset decimals.
Recommendations	Override the decimals function and make it return 18 if the vault is expected to work with tokens with multiple decimal tokens.
Comments / Resolution	Acknowledged.

Issue_58	Interest calculation is slightly underestimated
Severity	**Informational**
Description	The accrued interest calculated on _computeUpdatedAssets uses a binomial approximation to calculate compounded interest. However, this calculation always slightly underestimates the compounded interest. For example, when depositing 1 asset with 100% APY for a whole year, the real compounded interest should yield 2.71 but this approximation will result in 2.66.
Recommendations	Safe to acknowledge but document this behavior.
Comments / Resolution	Acknowledged.

Issue_59	Bad governance settings can lead to loss/under-collateralization
Severity	**Governance**
Description	The Savings contract pays out interest at a fixed rate to the participants of the savings system. This is done by minting stablecoin tokens directly to the Savings contract which can be claimed by the participants. Since stablecoins are minted with no backing, the system needs to be monitored closely to ensure proper functioning. Firstly, the system has to be profitable, so the amount paid out as interest should be lower than the fees collected through flash loans and swap fees. Secondly, tokens are minted uncollateralized at a fixed rate. So any user can suddenly deposit a massive amount of stablecoins to the savings system and make the system pay out far more interest than intended, leading to an undercollateralization of the system.

	To prevent this, the admins must do frequent checks of the health of the system to ensure the interest paid out doesn't degrade the system's health. The Angle protocol documentation presents a few key points to keep in mind when setting rates. Constant monitoring of the health of the system as a whole is important.
Recommendations	System health needs to be constantly monitored since the interest paid to the Savings contract is an expense to the system, and can lead to undercollateralization if not managed properly.
Comments / Resolution	Acknowledged.