BAIL security

Parallel Protocol Fee Claim Update

FINAL REPORT

January '2026

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 - Fee Claim Update - Audit Report
Website	app.parallel.best
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/parallel-protocol/parallel-parallelizer/tree/3fd5aa8a5a335d54e72d4dc167f19541a769c79b
Resolution 1	https://github.com/parallel-protocol/parallel-parallelizer/tree/20674ad9a3a82bdeb50f9bf2b8248dad13c872b1
Resolution 2	https://github.com/parallel-protocol/parallel-parallelizer/tree/2dfad6252bf84c3b1d66607f8f9969a164bb26ff

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no changes made)	Failed resolution	Open
High	2	2				
Medium	1	1				
Low	4	4				
Informational	8	6		2		
Governance						
Total	15	13		2		

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

Surplus Upgrade

This audit focuses on just an upgrade to the Parallelizer system involving the surplus mechanism. The previous audit on the Parallel V3 system as a whole can be found here, which covers all the contracts in the system. All acknowledged issues from the previous report are still present in the current codebase and have not been re-reported here for brevity.

This upgrade adds a surplus module that lets the protocol identify when it is holding "excess" collateral value, swap that excess into the stable asset under a configurable slippage limit, and then distribute the resulting income to a set of recipients according to predefined shares. The Parallelizer protocol accrues fees in terms of the collateral token during mints and burns, and this surplus module implements a way to collect and distribute these fees in terms of the stable token while still maintaining the collateralization of the protocol as a whole.

Surplus

The Surplus contract is a restricted "fee capture" module for the Parallelizer system. It allows an authorized operator to periodically skim excess collateral value that accumulates in the diamond (i.e., collateral held beyond what is required to back the outstanding stable supply) and convert that excess into the protocol's stablecoin. To do this, processSurplus computes the protocol's surplus for a given collateral using the system's oracle pricing and accounting, then swaps the surplus amount into the stable asset while enforcing a minimum-outbound derived from a configurable slippage tolerance; this turns unrealized surplus into realized stablecoin revenue held by the protocol.

Once surplus has been converted into stablecoins, the release function distributes the accumulated stablecoin balance to a predefined set of payees according to share weights. This provides an on-chain mechanism for routing protocol revenue to recipients (or burning, if configured) under access control, and emits events for monitoring when surplus is processed and when income is released.

Privileged Functions

- processSurplus
- release

Core Invariants:

INV 1: Surplus amount must be swapped into the tokenP stablecoin before being credited.

Issue_01	Surplus processing can depeg a healthy protocol
Severity	**High**
Description	processSurplus lets an authorized actor “skim” surplus from a single collateral and swap it into the stablecoin, effectively minting new stable liabilities backed by that collateral’s perceived excess value. The problem is that “surplus” is computed per-collateral, with no protocol-wide safety buffer. If one collateral depegs or drops in price, the system can still be barely healthy overall, but other collaterals may still show local surplus. Harvesting that local surplus increases total stable supply and can push the whole system into undercollateralization. Assume tokens C1, C2, and C3 are all stablecoin collaterals in the system. Say Parallelizer holds 100 tokens of each and has minted out 291 tokenP stablecoins. Each collateral token backs 97 tokenP mint. Collateralization Ratio = $300/$291 = 1.03 -> healthy Say C1 depegs to 0.95. CR = $(95+100+100)/$291 = 1.013 -> still healthy Now, say the admin calls processSurplus on tokens C2 and C3. There are 100 collateral tokens each, which back 97 tokenP each. So there's $3 extra each. So 3 tokenP is minted from each of C2 and C3. Now, CR = $(95+100+100)/$(291+3+3) = 0.993 -> unhealthy. So this allows the admin to undercollateralize a healthy system. This is a severe violation of the safeguards of the system, and can even get triggered by mistake, depending on market conditions.
Recommendations	Consider preventing surplus minting if the overall CR is below a specified value.

Comments / Resolution	Fixed by following recommendations.

Issue_02	No surplus buffer can lead to easy depegs due to oracle deviations
Severity	**Low**
Description	processSurplus treats any positive “surplus” as immediately extractable and converts it into newly issued stablecoins. There is no minimum global collateral ratio, safety margin, or “buffer” requirement before surplus can be harvested. Furthermore, the admins cannot even specify the amount of surplus to extract; they can only specify the collateral. This leads to a system that can very easily become undercollateralized due to small oracle price changes. The system mints tokenP until there's 0 excess collateral (ignoring mint fees). Thus, any slight price drop immediately after a mint will temporarily undercollateralize the protocol since there is exactly 0 room for any price movement.
Recommendations	Consider ensuring that the overall collateral ratio of the protocol is above a defined minimum after processing the surplus, and allowing the admins to specify how much surplus collateral to mint against.
Comments / Resolution	Fixed by following recommendations.

Issue_03	Stored USDp balance can be accidentally sold as rewards
Severity	**Low**
Description	The RewardHandler facet allows the trusted seller to swap any token not recognized as a PSM collateral, enabling conversion of protocol rewards into collateral tokens. With the new surplus mechanism, some USDp may remain in the PSM contract until released. This creates a risk where the trusted seller could unintentionally sell stored USDp, converting it back into collateral. While this action is reversible by processing the surplus again, it may result in minor value loss due to swap fees.
Recommendations	If governance is fully aware of this risk and the trusted seller is instructed to avoid selling USDp, the current behavior may be acknowledged. Otherwise, update sellRewards to explicitly prevent selling USDp.
Comments / Resolution	Fixed by following recommendations.

Issue_04	The surplus mechanism might be blocked due to collateral caps or pausability.
Severity	**Informational**
Description	Processing surplus relies on the ability to swap collateral tokens for USDp: *issuedAmount = ISwapper(address(this)).swapExactInput(* *collateralSurplus, minExpectedAmount, collateral,* *address(ts.tokenP), address(this), block.timestamp* *);* If minting is paused or the collateral cap has been reached, the swap cannot proceed, preventing surplus processing entirely. This may interfere with the intended operation of the surplus mechanism.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_05	Improper approval handling
Severity	**Informational**
Description	The processSurplus function grants approval to itself on the collateral tokens for minting tokenP. *IERC20(collateral).approve(address(this), collateralSurplus);* Even though SafeERC20 is available, forceApprove is not used. This can cause issues with non-standard tokens, which handle approvals differently.
Recommendations	Use forceApprove to handle approvals.
Comments / Resolution	Fixed by following recommendations.

Issue_06	DoS on processSurplus if the collateral token reverts on self-transfers.
Severity	**Informational**
Description	The processSurplus function calls swapExactInput to convert surplus collateral into USDp. During execution, the _swap function performs a token transfer: *IERC20(tokenIn).safeTransferFrom(msg.sender, address(this), amountIn);* In this context, the caller is the contract itself, resulting in a self-transfer. While most tokens allow this, some revert when the sender and receiver are the same. For example, Compound V3 tokens like cUSDCv3 exhibit this behavior. If such tokens are accepted as collateral in the PSM in the future, this would break the surplus mechanism for those assets.
Recommendations	Either acknowledge this behavior as an accepted limitation or update the swapper facet to skip the transfer when the caller is the PSM.
Comments / Resolution	Acknowledged.

LibSurplus

LibSurplus is the accounting and payout library backing the Surplus module. It computes a collateral's "excess" by valuing the protocol's on-contract collateral balance via the configured oracle and comparing it to the protocol's recorded stable liabilities, then converts that value surplus into a collateral amount to be swapped. After surplus has been realized as stablecoin revenue, release splits the income across a configured set of payees using share weights (optionally burning the portion assigned to the zero address) and emits events for distribution tracking. It also provides a helper to derive a minimum acceptable swap output from a slippage tolerance, allowing surplus conversions to be bounded against adverse execution.

Core Invariants:

INV 1: Surplus must only report the excess collateral, the part that does not back any of the minted stablecoins yet.
INV 2: During release, each target must receive tokens proportional to their share amount.

Issue_07	Incorrect surplus calculation
Severity	**High**
Description	LibSurplus subtracts a normalized debt value as if it were the actual stable supply. *stableSurplus = totalCollateralValue - collatInfo.normalizedStables;* But collatInfo.normalizedStables is the normalized value, which remains unchanged during redemptions. For the actual amount of stablecoins minted against it, the value needs to be denormalized using ts.normalizer. Thus, this calculation is only correct when ts.normalizer=1e27, the unit value. At all other values, this is incorrect.
Recommendations	De-normalize the value and then use it.
Comments / Resolution	Fixed by following recommendations.

Issue_08	Surplus calculation doesn't work for managed collaterals
Severity	**Medium**
Description	_computeCollateralSurplus uses balanceOf to calculate the collateral amount and ignores the existence of managed collaterals. Parallelizer supports managed collaterals, where the collateral tokens are sent off to a different contract to be invested to generate yield. For such managed collaterals, balanceOf will give the wrong value since the collateral tokens do not exist in the Parallelizer contract itself.
Recommendations	For managed collaterals, use LibManager.totalAssets to get the collateral amount present in the system. These funds also need to be released first before they can be minted against.
Comments / Resolution	Fixed following recommendations.

Issue_09	Revert message for code clarity
Severity	**Informational**
Description	The stableSurplus value is calculated using a raw subtraction. *stableSurplus = totalCollateralValue - collatInfo.normalizedStables;* If there isn't any surplus, this just reverts with an underflow. Consider adding a check and reverting with an appropriate custom error/message.
Recommendations	Consider adding a custom error/message when the stableSurplus calculation underflows.
Comments / Resolution	Fixed by following recommendations.

Getters

The Getters contract introduced a few new functions in this protocol upgrade. The getPayees, getTotalShares, getShares, getSlippageTolerance, getLastReleasedAt, getCollateralSurplus functions are simple view functions that fetch the new storage variables introduced in the upgrade.

No issues were identified in this contract.

Setters

The Setters contract and the LibSetters library introduce the updatePayees and updateSlippageTolerance functions, which enable the Governor to specify the address to which the surplus mints are to be directed and the tolerance for mints.

Issue_10	Payee updates can be blocked by a failing income release
Severity	**Low**
Description	updatePayees attempts to distribute any accumulated protocol income to the current payees before applying the new payee configuration. If that distribution fails (for example, because the "burn" route is configured via the zero address but burning is disabled, or because the stable token transfer/burn path reverts for any reason), the whole updatePayees call reverts. This creates a governance footgun: the protocol can get stuck in a state where it cannot change payees precisely because the old payee configuration is broken, turning a recoverable configuration error into a persistent inability to update revenue routing.
Recommendations	Consider adding logic to allow payee updates without a surplus release, or documenting and acknowledging this issue.
Comments / Resolution	Fixed by following recommendations.

Issue_11	Duplicate payees can lead to incorrect share accounting
Severity	**Low**
Description	updatePayees does not prevent duplicate addresses in the payee list, but it stores shares in a mapping keyed by address while also iterating over the payees array during release. If a payee appears twice, the mapping only retains the last share value, yet release will pay that same address multiple times using the same (last) share each time. Example: set payees to [Alice, Alice] with shares [1, 100]; totalShares becomes 101, shares[Alice] becomes 100, and release will attempt to pay Alice about $99.01 twice from a $100 income, which can overdraw the balance and revert, effectively bricking income distribution.
Recommendations	Consider adding a duplicate check in the updatePayees function.
Comments / Resolution	Fixed by following recommendations.

Issue_12	No event emission on updateSlippageTolerance
Severity	**Informational**
Description	Currently, the function updateSlippageTolerance does not emit an event when slippage tolerance is changed for a collateral. *function updateSlippageTolerance(address collateral, uint256 slippageTolerance) internal {* *if (slippageTolerance > BASE_9) revert InvalidRate();* *ParallelizerStorage storage ts = s.transmuterStorage();* *ts.slippageTolerance[collateral] = slippageTolerance;* It is considered a best practice to emit events when state variables are updated.
Recommendations	Consider emitting events when slippage is updated.
Comments / Resolution	Fixed by following recommendations.

Issue_13	updatePayees can accidentally transfer USDp to a compromised address.
Severity	**Informational**
Description	The updatePayees function calls LibSurplus.release when there is a non-zero USDp balance and the list of previous payees is not empty: *if (income > 0 && ts.payees.length > 0) {* *LibSurplus.release(income, ts.payees);* *}* This behavior can lead to unintended asset loss if any of the previous payees have been compromised. In such cases, executing updatePayees would transfer USDp to addresses that may no longer be trusted.
Recommendations	Consider adding a boolean flag to updatePayees to allow the caller to skip the release call when needed, or acknowledge this issue.
Comments / Resolution	Fixed by following recommendations.

Issue_14	Stale shares mapping after payee removal
Severity	**Informational**
Description	When updatePayees is called, the payees array is deleted, but individual entries in the shares mapping are not cleared. Old payees retain non-zero share values even though they are no longer active. This delete statement removes the payees array but does not clear the shares mapping for the removed addresses. As a result, the getShares getter will return stale non-zero values for addresses that were previously payees but have since been removed.
Recommendations	Consider zeroing out old payee values.
Comments / Resolution	Fixed by following recommendations.

Storage

The Storage contract introduces a few extra slots in the Parallelizer storage to hold data related to the surplus system.

No issues were identified in this contract.

Swapper

This contract contains a small modification in this upgrade, which rearranges the order of operations in the _swap function.

Issue_15	Hard cap check uses different rounding in quotes vs. execution
Severity	**Informational**
Description	In the _swap function, the mint path enforces the per-collateral stablecoin cap via _checkHardCaps, but the cap check is performed on different values in the quoting functions versus the actual swap execution. quoteIn / quoteOut check caps using the user-facing amountOut directly, while _swap first converts amountOut into a "normalized" amount using rounding (round up), updates storage, and then checks caps using the normalized accounting (effectively re-rounded when converted back). This rounding mismatch can create a 1-wei discrepancy. Swap: *uint128 changeAmount = (amountOut.mulDiv(BASE_27, ts.normalizer, Math.Rounding.Ceil)).toUint128();* *collatInfo.normalizedStables = collatInfo.normalizedStables + uint216(changeAmount);* *_checkHardCaps(collatInfo, 0, ts.normalizer);* Quote: *amountOut = _quoteMintExactInput(collatInfo, amountIn);* *_checkHardCaps(collatInfo, amountOut, ts.normalizer);* Swap first normalizes the amount, rounding up, and then checks the cap, rounding down, while the quote function does the exact amount. This discrepancy can lead to different results from the 2 functions.
Recommendations	Consider using the same sequence of rounding/checking in the

	quote and swap functions.
Comments / Resolution	Fixed by following recommendations.