MixBytes()

Diffuse Prime Security Audit Report

DECEMBER 22, 2025

Table of Contents

1. Introduction

1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

1.2 Executive Summary

Diffuse Prime is a non-custodial lending protocol that matches lender capital with borrower strategies. Vaults implement the ERC-4626 interface for easy integration, but they don't rely on the usual share-price mechanics: lender returns are tracked per strategy with a simple points/debt counter. Risk is isolated per vault. In v1, borrowers can post either the vault's base token (OriginalAsset) or the strategy token (StrategyAsset, e.g., Pendle PT) as collateral. A small set of curators decides which strategies are allowed and sets fees/limits via a timelock, per vault.

SGX secures the most sensitive steps. When a borrow request is activated or a position must be liquidated, a secure enclave computes prices and checks, then submits a proof that the blockchain verifies. Only after verification does the vault execute: if risk limits are breached, the position is unwound along the reverse route, lender principal and interest are paid first, and any shortfall is recorded as a vault-local deficit.

During the audit, the following attack vectors were checked:

1. We verified that a borrower can remove their request only until the deadline; after a position is registered, `removeBorrowRequest()` reverts in `removePendingBorrowerPosition()` because the position is active, not pending.
2. We confirmed that the project does not support fee-on-transfer tokens: there are before/after balance checks around transfers and an additional `require` ensuring the transferred amount equals the expected amount.
3. We verified that `totalLenderAssetsLocked` during `removeBorrowRequest()` is decreased by the same value that was used during `createBorrowRequest()`; there is no mismatch and no state inconsistency.
4. We checked that all routes from external protocols are added by whitelisted curators during strategy creation, so no malicious routes can be used.
5. We confirmed that `currentAmount` is properly updated after each adapter swap, ensuring subsequent swaps in the route use the actual output amount from previous swaps rather than the original input amount.
6. We verified that the diamond proxy pattern is correctly implemented in `VaultProxy`: all external functions from all four facets (`BaseFacet`, `BorrowFacet`, `LiquidationFacet`, `ERC4626Facet`) are properly enumerated in `VaultProxySelectors.getImplementation()`, and delegatecalls are made only to whitelisted facets.
7. We verified that borrowers cannot bypass SGX attestation by creating a pending borrow request and immediately calling `unborrow()`. The call fails at

BorrowStorageLib.finishActiveBorrowerPosition() because it only works on active positions, causing a revert that rolls back all token transfers.
8. We developed and ran integration tests on a mainnet fork to exercise most end-to-end scenarios and validate critical invariants, ensuring real-world correctness under production-like conditions.

The audit scope included review of the BorrowLogicLib logic that handles asynchronous adapter interactions (e.g., unfinished swaps, cached routes, continuation flows). However, the actual asynchronous adapter implementations were not provided in the scope, and their internal logic (e.g., _continueSwap() implementations, cooldown mechanisms, multi-step external protocol interactions) was outside the audit scope. All adapters provided in the scope operate synchronously and complete their operations within a single transaction. Future implementations of asynchronous adapters should undergo additional security review to ensure proper handling of incomplete swap states, route management during multi-step operations, and emergency recovery mechanisms.

1.3 Project Overview

Summary

Title	Description
Client Name	Diffuse
Project Name	Prime
Type	Solidity
Platform	EVM
Timeline	25.09.2025 - 18.12.2025

Scope of Audit

File	Link
src/facets/BaseFacet.sol	BaseFacet.sol
src/facets/LiquidationFacet.sol	LiquidationFacet.sol
src/facets/ERC4626Facet.sol	ERC4626Facet.sol
src/facets/BorrowFacet.sol	BorrowFacet.sol
src/vault-registry/VaultRegistry.sol	VaultRegistry.sol

File	Link
src/external/PendleLib.sol	PendleLib.sol
src/libs/ErrorLib.sol	ErrorLib.sol
src/libs/BorrowStorageLib.sol	BorrowStorageLib.sol
src/libs/BaseLogicLib.sol	BaseLogicLib.sol
src/libs/BorrowLogicLib.sol	BorrowLogicLib.sol
src/libs/OZStorageLib.sol	OZStorageLib.sol
src/libs/EventLib.sol	EventLib.sol
src/libs/TypeLib.sol	TypeLib.sol
src/libs/BaseStorageLib.sol	BaseStorageLib.sol
src/adapters/Adapter.sol	Adapter.sol
src/adapters/PendleAdapter.sol	PendleAdapter.sol
src/adapters/CurveStableSwapAdapter.sol	CurveStableSwapAdapter.sol
src/HelperMethods.sol	HelperMethods.sol
src/VaultProxySelectors.sol	VaultProxySelectors.sol
src/Common.sol	Common.sol
src/VaultProxy.sol	VaultProxy.sol
src/adapters/AdapterWithWhitelist.sol	AdapterWithWhitelist.sol

Versions Log

Date	Commit Hash	Note
25.09.2025	e4bf94812ff041e63dd92262b17e767ec5c286c4	Initial Commit
08.12.2025	0350a678104d9985181f8295e56d75137f3daaa7	Re-audit Commit

Date	Commit Hash	Note
18.12.2025	ddcd3f50ba065c2acff0bd0be3b1bc4b69c5478a	Re-audit Commit

Mainnet Deployments

File	Address	Blockchain
VaultRegistry.sol	0x1F6D4EEd083634020E2d0f11b3d401c28fD962c6	Ethereum
BaseFacet.sol	0x38866469197FF68854404A7BE8C52e6Acd63beE2	Ethereum
BorrowFacet.sol	0xe9393bEb9DC21959BE859e14FB7b4B1071fB94a1	Ethereum
LiquidationFacet.sol	0xaFb922CeE2691D3697d86344F96B6a48b4Cb2d24	Ethereum
ERC4626Facet.sol	0x2AF8AbDC28390e377Ba46ea8c5b53066CBE535e0	Ethereum
VaultProxy.sol	0x42C7E24d974A8790737F4f2237F60C2F93Cacf64	Ethereum

1.4 Security Assessment Methodology

Project Flow

Stage	Scope of Work
Interim audit	**Project Architecture Review:** · Review project documentation · Conduct a general code review · Perform reverse engineering to analyze the project's architecture based solely on the source code · Develop an independent perspective on the project's architecture · Identify any logical flaws in the design OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS.
	Code Review with a Hacker Mindset: · Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities. · Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead. · Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws. · Review test cases and in-code comments to identify potential weaknesses. OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY.
	Code Review with a Nerd Mindset: · Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits. · Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors. OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS.

Stage	Scope of Work
	Consolidation of Auditors' Reports: · Cross-check findings among auditors · Discuss identified issues · Issue an interim audit report for client review OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT.
Re-audit	**Bug Fixing & Re-Audit:** · The client addresses the identified issues and provides feedback · Auditors verify the fixes and update their statuses with supporting evidence · A re-audit report is generated and shared with the client OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED.
Final audit	**Final Code Verification & Public Audit Report:** · Verify the final code version against recommendations and their statuses · Check deployed contracts for correct initialization parameters · Confirm that the deployed code matches the audited version · Issue a public audit report, published on our official GitHub repository · Announce the successful audit on our official X account OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT.

1.5 Risk Classification

Severity Level Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likehood: High	Critical	High	Medium
Likehood: Medium	High	Medium	Low
Likehood: Low	Medium	Low	Low

Impact

- **High** – Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** – Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** – One-time contract lock that can be fixed by the administrator without a contract upgrade.

Likelihood

- **High** – The event has a 50–60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** – An unlikely event (10–20% probability of occurring) that can be triggered by a trusted actor.
- **Low** – A highly unlikely event that can only be triggered by the owner.

Action Required

- **Critical** – Must be fixed as soon as possible.
- **High** – Strongly advised to be fixed to minimize potential risks.
- **Medium** – Recommended to be fixed to enhance security and stability.
- **Low** – Recommended to be fixed to improve overall robustness and effectiveness.

Finding Status

- **Fixed** – The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** – The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** – The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

1.6 Summary of Findings

Findings Count

Severity	Count
Critical	0
High	2
Medium	7
Low	14

Findings Statuses

ID	Finding	Severity	Status
H-1	Pendle Adapter Uses Wrong `tokenOut` in `sell` Path Causing `unborrow()` and `liquidate()` to Revert	High	Fixed
H-2	Whale Yield Theft Attack via Share Dilution	High	Fixed
M-1	SGX Proof Length Mismatch Blocks All Liquidations and Borrower Activations	Medium	Fixed
M-2	Partial Yield Loss on Insufficient Vault Balance	Medium	Fixed
M-3	SGX Attestation Fee Risk Blocks Borrower Activations and Liquidations	Medium	Fixed
M-4	Async Adapters Lock In-Flight Swaps to Old Adapter and Route	Medium	Fixed
M-5	Incorrect End Date Check Logic for Curator in Unborrow Function	Medium	Fixed
M-6	Liquidation Price Validation Always Passes Due to Incorrect Order of Operations	Medium	Fixed
M-7	Async Unfinished Swap Cannot Be Reused for Multiple Rounds	Medium	Fixed

L-1	Unfair Withdrawal Distribution During Base Asset Deficit	Low	Acknowledged
L-2	Inaccurate NatSpec Documentation for `liquidate()` Function	Low	Fixed
L-3	Unnecessary End Date Check in Strategy Loop	Low	Fixed
L-4	Inconsistent Comparison Operator in Strategy End Date Check	Low	Fixed
L-5	Unused Function `setBorrowApr()` in `BaseStorageLib`	Low	Fixed
L-6	`MIN_COLLATERAL_GIVEN` Check Inconsistent Across Token Decimals	Low	Fixed
L-7	`delegatecall` Return Value Not Checked in Adapter Emergency Function	Low	Fixed
L-8	Vault Removal Breaks Liquidation System	Low	Fixed
L-9	Hardcoded Curve Pool Token Indexes Limit Adapter Flexibility	Low	Fixed
L-10	Race Condition: `minAssetsOut` Denomination May Mismatch After Route Update	Low	Fixed
L-11	Curve Adapter Accepts Identical Token Indexes	Low	Fixed
L-12	Permissive Default Whitelist in Adapters Allows Any Caller	Low	Fixed
L-13	Liquidation Request Event Includes Skipped Positions	Low	Fixed
L-14	Unreachable Code in `Adapter.continueSwap()`	Low	Fixed

2. Findings Report

2.1 Critical

Not Found

2.2 High

H-1	Pendle Adapter Uses Wrong tokenOut in sell Path Causing unborrow() and liquidate() to Revert		
Severity	High	**Status**	Fixed in 0350a678

Description

The PendleAdapter._sell() implementation passes TOKEN_OUT as the tokenOut parameter to createTokenOutputSimple(...) when calling swapExactPtForToken. In the protocol design, TOKEN_OUT represents the position asset (PT), while the sell path intends to unwind PT back to the base asset (TOKEN_IN, e.g., WETH). Supplying PT as tokenOut makes the SY redemption attempt invalid and reverts with SYInvalidTokenOut.

Observed code in src/adapters/PendleAdapter.sol:

```
function _sell(uint256 amount) internal override {
    TOKEN_OUT.forceApprove(POOL, amount);
    IPActionSwapPTV3(POOL).swapExactPtForToken({
        receiver: address(this),
        market: MARKET,
        exactPtIn: amount,
        // wrong: PT is not a valid redeem token
        output: createTokenOutputSimple(address(TOKEN_OUT), 0),
        limit: createEmptyLimitOrderData()
    });
}
```

User impact scenarios where this revert will occur:

- Unborrow flow: when closing a borrower position and unwinding PT back to the vault's base asset, the sell path will revert, preventing the position from exiting normally.
- Liquidations: when the system tries to convert PT to base asset during liquidation settlement, the conversion reverts, blocking liquidation proceeds distribution.

Recommendation

We recommend replacing TOKEN_OUT with TOKEN_IN in the sell path so that Pendle redeems SY to the correct base asset:

```
output: createTokenOutputSimple(address(TOKEN_IN), 0)
```

Client's Commentary:

✓ This is fixed: PR-31 and PR-31

H-2	Whale Yield Theft Attack via Share Dilution		
Severity	High	**Status**	Fixed in 0350a678

Description

The yield distribution mechanism in BaseLogicLib.sol contains a vulnerability that allows large depositors (whales) to steal yield from existing lenders during periods when their funds are not utilized by borrowers. The issue stems from the yield distribution formula that divides yield among all shareholders, including those whose funds are not actually utilized by borrowers.

The yield calculation in updateLenderPoints() distributes yield based on total shares rather than utilized shares:

```
uint256 yieldInIntervalInPrecisionDividedByTotalShares =
    assetsUtilized * borrowApr * timeDelta * PRECISION_IN_LENDER_POINTS
        / (APR_MULTIPLIER * YEAR * totalShares);
```

Where:

- assetsUtilized = only funds actually borrowed by borrowers
- totalShares = all shares in the vault (including non-utilized funds)

Attack Scenario:

1. Two honest lenders deposit 100k USDC each (total 200k USDC)
2. Borrowers utilize all 200k USDC at 25% APR, generating 50k USDC annual yield
3. Whale deposits 1M USDC (funds not utilized by borrowers initially)
4. Whale's funds remain unused for 1 week before borrowers request them
5. During this week, yield accumulates based on diluted formula (1M / 1.2M = 83.33% share)
6. After 1 week, borrowers utilize whale's 1M USDC
7. Whale can frontrun future borrow requests and withdraw/re-deposit to repeat the attack
8. Whale receives disproportionate yield during unused periods despite not taking lending risk

This issue is classified as **High** severity because it allows whales to extract disproportionate yield during unused fund periods through timing manipulation, creating unfair distribution that penalizes honest lenders.

Recommendation

We recommend implementing yield distribution based on utilized shares only, ensuring that only lenders whose funds are actually borrowed by borrowers receive yield during those periods.

Client's Commentary

We've added white list of lenders to prevent malicious lenders from participating. And we also will recomend all borrowers to use MEV-protection at all times. In general, for an attacker to execute this type of exploit, the lender must be confident they can withdraw their assets afterward. However, this is not guaranteed: other lenders may frontrun the withdrawal, causing the attacker's assets to remain locked in the borrower's position. So in reality, this attack is feasible only if the lender holds a major portion of the vault. But if it is so, then the utilization of the vault is close to 50% or lower and the profit per lender share is significantly lower than the target APY. Which makes the attack economically unsustainable.

2.3 Medium

M-1	SGX Proof Length Mismatch Blocks All Liquidations and Borrower Activations		
Severity	Medium	**Status**	Fixed in 0350a678

Description

The SGX_PROOF_SIZE constant is set to 397 bytes, but the actual output from IAutomataDcapAttestationFee.verifyAndAttestOnChain() returns 461 bytes. This causes all SGX operations (borrower position activation and liquidations) to fail with InvalidOutputLength error, completely breaking the protocol's core functionality.

The issue occurs in BorrowLogicLib.validateProof():

```
require(output.length == SGX_PROOF_SIZE,
        ErrorLib.InvalidOutputLength(output.length)
);
```

This occurs because it's assumed that advisoryIDs: new string[](0) is encoded as 0 bytes, but it's actually encoded as 64 bytes (32 bytes for offset and 32 bytes for length). V3QuoteVerifier.sol#L160

This is the test that shows that output length differs from 397.

To run the test:

1. Put the test file into the ./test folder
2. Use forge t --mt testSGXProofSize -vvvv

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";

/**
 * @title OutputEncodingTest
 * @notice This test proves that SGX_PROOF_SIZE should be 461 bytes
 * @dev This demonstrates the actual output length from serializeOutput function
 */
contract OutputEncodingTest is Test {
    enum TCBStatus {
        OK,
        TCB_SW_HARDENING_NEEDED,
        TCB_CONFIGURATION_AND_SW_HARDENING_NEEDED,
        TCB_CONFIGURATION_NEEDED,
        TCB_OUT_OF_DATE,
        TCB_OUT_OF_DATE_CONFIGURATION_NEEDED,
        TCB_REVOKED,
        TCB_UNRECOGNIZED
    }

    struct Output {
        uint16 quoteVersion; // BE
        bytes4 tee; // BE
        TCBStatus tcbStatus;
        bytes6 fmspcBytes;
        bytes quoteBody;
        string[] advisoryIDs;
    }

    /**
     * @notice Exact copy of serializeOutput function from QuoteVerifierBase.sol
     * @dev This is the actual function used in the audited code
     */
    function serializeOutput(Output memory output)
        internal
        pure
        returns (bytes memory)
    {
        return abi.encodePacked(
            output.quoteVersion,
            output.tee,
            output.tcbStatus,
            output.fmspcBytes,
            output.quoteBody,
            abi.encode(output.advisoryIDs)
        );
    }
```

```
    /**
     * @notice Test that proves the actual output length is 461 bytes
     * @dev This test replicates the exact serializeOutput function behavior
     */
    function testSGXProofSize() public {
        // Create arbitrary 384-byte array (simulating EnclaveReport)
        bytes memory arbitrary384Bytes = new bytes(384);

        // Fill with some pattern for testing
        for (uint i = 0; i < 384; i++) {
            arbitrary384Bytes[i] = bytes1(uint8(i % 256));
        }

        assertEq(arbitrary384Bytes.length, 384);

        // Create Output struct exactly like V3QuoteVerifier does
        Output memory output = Output({
            quoteVersion: 3,
            tee: 0x00000000, // SGX_TEE
            tcbStatus: TCBStatus.OK,
            fmspcBytes: bytes6(0x123456789abc),
            quoteBody: arbitrary384Bytes, // 384 bytes (ENCLAVE_REPORT_LENGTH)
            advisoryIDs: new string[](0) // Empty array (like in V3)
        });

        bytes memory serialized = serializeOutput(output);

        assertEq(serialized.length, 461);
    }
}
```

This issue is classified as **Medium** severity because it blocks core protocol functionality, specifically borrower position activations and liquidations, making the protocol completely non-functional for SGX-dependent features.

Recommendation

We recommend applying the following:

1. Update `SGX_PROOF_SIZE` constant to 461 bytes
2. Add integration tests with real SGX proofs to validate output length
3. Update mocked test data to match real SGX output structure

Client's Commentary

Actual serializeOutput has `output.advisoryIDs.length > 0 ? abi.encode(output.advisoryIDs) : bytes("")`. PR-31 edited comment with link to automata (changed commit)

M-2	Partial Yield Loss on Insufficient Vault Balance		
Severity	Medium	**Status**	Fixed in 0350a678

Description

`BaseLogicLib.withdrawYield()` attempts to pay lenders their accrued yield. The vulnerable path is reachable only when the vault's raw ERC-20 balance is lower than the yield owed – a situation that happens after a borrower under-repays and `baseAssetDeficit` is recorded. When the deficit exists (`remainedBalance < unpaidYield + pending`), the function caps the transfer with:

```
if (yield > remainedBalance) {
    yield = remainedBalance;
}
```

After that, it conditionally updates state only if `yield > 0`:

```
if (yield > 0) {
    BaseStorageLib.decreaseUnpaidYield(user, unpaidYield, strategyId);
    uint256 newDebt =
        userShares * accPointsPerShare / PRECISION_IN_LENDER_POINTS;
    BaseStorageLib.setLenderPointsDebt(user, newDebt, strategyId);
    BaseStorageLib.setTotalDebt(
        BaseStorageLib.getTotalDebt(strategyId) + newDebt - userDebt,
        strategyId
    );
    withdrawnYield += yield;
}
```

When the vault balance is positive but still lower than the total owed, the whole `unpaidYield` value is deleted even though only a part of it is really paid. The remaining (unpaid) portion disappears from storage because:

1. `decreaseUnpaidYield` burns the full `unpaidYield` value.
2. `userDebt` is bumped to the latest index, so the unpaid part no longer appears in `pending` on the next call.

Consequently, the lender permanently loses the difference between what was owed and what was actually transferred.

Here's an example:

1. Lender A and Lender B each supply `1,000 USDC`. APR = `25%`, each expects `250 USDC` yield.
2. Borrower is liquidated; only `2,400 USDC` returned, `baseAssetDeficit` is increased to `100`.
3. Vault balance before withdrawals: `2,400 USDC`.
4. Lender A calls `withdraw(1,000)` – vault sends `1,000 USDC` principal, vault's balance now `1,400 USDC`.
5. Lender B calls `withdraw(1,000)` again – vault sends another 1,000 USDC, vault's balance now `400 USDC`.

6. Lender A calls withdrawYield() – owes 250 USDC, vault has 400 USDC, so pays 250 (balance now 150 USDC).
7. Lender B calls withdrawYield() – owes 250 USDC, but remainedBalance = 150, so pays only 150. Entire unpaidYield (250) is cleared from storage, but only 150 was paid. Lender B permanently loses 100 USDC yield.

This issue is classified as **Medium** severity because it requires a deficit event but, once triggered, causes irreversible loss of lender yield, silent accounting drift, and unfair *first-come-first-served* payouts, while leaving principal ultimately recoverable.

Recommendation

We recommend modifying the BaseLogicLib.withdrawYield() function to preserve unpaid yield when vault balance is insufficient. The current implementation incorrectly clears the entire unpaidYield amount from storage even when only partial payment is made.
This ensures that lenders can claim their remaining yield when vault balance increases in the future, preventing permanent loss of earned yield due to insufficient vault balance.

Client's Commentary

✓ *PR-31 yield calculating, did small refactor not to repeat code for yield calculating*

M-3	SGX Attestation Fee Risk Blocks Borrower Activations and Liquidations		
Severity	Medium	**Status**	Fixed in 0350a678

Description

The `IAutomataDcapAttestationFee.verifyAndAttestOnChain()` function in the SGX attestation contract has a `collectFee()` modifier that can charge fees based on gas consumption.
The `collectFee()` modifier implementation:

```
    modifier collectFee() {
        uint256 txFee;
        if (_feeBP > 0) {
            uint256 gasBefore = gasleft();
            _;
            uint256 gasAfter = gasleft();
            txFee = ((gasBefore - gasAfter) * tx.gasprice * _feeBP) / MAX_BP;
            if (msg.value < txFee) {
                revert Insuccifient_Funds();
            }
        } else {
            _;
        }

        // refund excess
        if (msg.value > 0) {
            uint256 excess = msg.value - txFee;
            if (excess > 0) {
                // refund the sender, rather than the caller
                // @dev may fail subsequent call(s), if the caller were a
                // contract that might need to make subsequent calls
                // requiring ETH transfers
                _refund(tx.origin, excess);
            }
        }

    }
```

Currently, the `_feeBP` is set to 0, but the contract owner can change it in the future.
If a non-zero value is set, all calls to `verifyAndAttestOnChain()` will fail.

```
(bool success, bytes memory output) =
    BaseStorageLib.getAttestationContract().verifyAndAttestOnChain(proof);

if (!success) {
    revert ErrorLib.AttestationFailed(output);
}
```

As a result, new borrower requests cannot be activated, and what is more dangerous, old active positions cannot be liquidated.

Recommendation
We recommend implementing a mechanism to send ETH with attestation calls when fees are enabled.

Client's Commentary
✓ Payability PR-31

M-4	Async Adapters Lock In-Flight Swaps to Old Adapter and Route		
Severity	Medium	**Status**	Fixed in ddcd3f50

Description

Synchronous adapters can effectively be "replaced" by updating the route. However, for asynchronous swaps that were intercepted (unfinished), continuation is hard-bound to the originally intercepted adapter address. The per-position `cachedRoute` stores this adapter and is not updatable for already in-flight positions, so even if the curator updates the global strategy route, affected positions will still be forced to continue through the old, potentially faulty adapter.

If that adapter contains a bug or its upstream integration changes, the async swap may get stuck or behave incorrectly, and the curator cannot swap out the adapter for that specific position by changing `cachedRoute`. In such a scenario, the only recovery path is off-chain/manual intervention (withdrawing funds from the adapter and redistributing them between borrower and lenders according to some ad-hoc logic). This breaks the invariant that the on-chain position state fully reflects reality and can lead to accounting drift between the position's recorded state and the actual asset distribution.

This issue is classified as **Medium** severity because it can lead to inconsistent positions that require manual rescue procedures, increasing operational risk and the chance of misallocation during recovery.

Recommendation

We recommend revisiting the async flow to allow safe adapter replacement for unfinished swaps.

Client's Commentary

1. *All adapters are made upgradable with 3-day wait period PR-41*
2. *Cached route is used only when a new route without the adapter with interception was set to strategy. In this case funds are stuck in such adapter (most likely in corresponding protocol), so such upgreadability is only decision*

M-5	Incorrect End Date Check Logic for Curator in Unborrow Function		
Severity	Medium	**Status**	Fixed in ddcd3f50

Description

In BorrowLogicLib.unborrow(), the end date check logic for curators contradicts the comment.

```
if (
    isCurator
    && !isPositionOwner
    && unfinishedSwap.interceptedOnAdapter != address(0)
) {
    // curator can unborrow (on behalf of borrower) if strategy is finished
    // or there is an unfinished swap
    TypeLib.StrategyData storage strategy =
        BaseStorageLib.getStorage().strategies[position.strategyId];
    require(
        strategy.endDate < block.timestamp,
        ErrorLib.StrategyNotFinished()
    );
}
```

Comment states that curator can unborrow if strategy is finished **OR** there is an unfinished swap
However, the logic is inverted. The endDate check executes only when unfinishedSwap.interceptedOnAdapter != address(0) (there is an unfinished swap), but according to the comment, it should execute when there is no unfinished swap.
Since currently all adapters are synchronous, unfinishedSwap.interceptedOnAdapter is always address(0), so the condition is always false and the endDate check is never executed.
Curators and oracles can call unborrow() at any time, even before endDate.
If an unfinishedSwap exists, curators would be forced to wait until strategy endDate to continue the swap, blocking recovery of funds stuck in adapters.
This issue is classified as **Medium** severity because curators can currently bypass the endDate restriction, and the inverted logic will prevent proper handling of async operations in the future.

Recommendation

We recommend fixing the condition to check endDate when there is **no** unfinished swap by changing the condition from unfinishedSwap.interceptedOnAdapter != address(0) to unfinishedSwap.interceptedOnAdapter == address(0).

Client's Commentary

✓ Fixed PR-42

M-6	Liquidation Price Validation Always Passes Due to Incorrect Order of Operations		
Severity	Medium	**Status**	Fixed in ddcd3f50

Description

In `BorrowLogicLib.sgxActivateBorrowerPosition()`, the liquidation price validation is performed after the position's `liquidationPrice` has already been updated, causing the validation to always pass.

The validation should compare the new `liquidationPrice` parameter with the original value stored in the position. However, `activatePosition()` internally calls `BorrowStorageLib.activateBorrowerPosition()`, which immediately updates `position.liquidationPrice` to the new value before the validation code executes.

Since `position.liquidationPrice` has already been updated to match the `liquidationPrice` parameter, `liquidationPriceDelta` is always `0`, `liquidationPriceDiff` is always `0`, and the validation always passes regardless of how much the new price differs from the original.

This effectively disables the liquidation price difference validation, allowing oracle to provide any `liquidationPrice` value without being constrained by `BaseStorageLib.getLiquidationPriceDiff()`, breaking the intended security mechanism.

This issue is classified as **Medium** severity because it disables an important validation mechanism that should protect against oracle manipulation or errors, potentially allowing positions to be activated with incorrect liquidation prices that could lead to unfair liquidations or borrower losses.

Recommendation

We recommend moving the liquidation price validation before calling `activatePosition()`, so it compares the new `liquidationPrice` parameter with the original value stored in the position before it gets updated.

Client's Commentary

✓ *Fixed PR-42*

M-7	Async Unfinished Swap Cannot Be Reused for Multiple Rounds		
Severity	Medium	**Status**	Fixed in ddcd3f50

Description

In `BorrowLogicLib.swap()`, the logic for handling asynchronous adapters does not support multiple continuation rounds on the same adapter without reverting.

```
if (!finished) {
    // * same adapter is intercepting
    BorrowStorageLib.addUnfinishedSwap({
        positionId: args.positionId,
        interceptedOnAdapter: interceptedOnAdapter,
        acquiredTokenB: currentAmount,
        user: unfinishedSwap.user
    });
    BorrowStorageLib.setCachedRoute(args.positionId, args.route);
    return (new uint256[](args.minAssetsOut.length), false);
}
```

When an adapter returns `finished = false` from `continueSwap()`, the code attempts to re-add an unfinished swap. However, `BorrowStorageLib.addUnfinishedSwap()` requires that no unfinished swap exists for the position:

```
function addUnfinishedSwap(...) internal {
    require(
        $.unfinishedSwaps[positionId].interceptedOnAdapter == address(0),
        ErrorLib.UnfinishedSwapAlreadyExists(positionId)
    );
    ...
}
```

Since an unfinished swap already exists from the previous call, the second and subsequent calls where `continueSwap()` returns `finished = false` will revert with `ErrorLib.UnfinishedSwapAlreadyExists(positionId)` instead of updating the existing unfinished swap state.
This issue is classified as **Medium** severity because when async adapters are added, positions will become stuck after the first continuation step, requiring manual intervention or protocol upgrades to recover funds.

Recommendation

We recommend modifying the unfinished swap handling logic to update the existing unfinished swap instead of trying to add a new one on subsequent `continueSwap()` calls.

Client's Commentary

✓ *Fixed PR-42*

2.4 Low

L-1	Unfair Withdrawal Distribution During Base Asset Deficit		
Severity	Low	**Status**	Acknowledged

Description

When `baseAssetDeficit` is greater than zero, the vault becomes technically insolvent, yet the withdrawal mechanism creates unfair distribution among existing lenders. The `getMaxWithdraw()` function limits withdrawals by vault balance, creating a *first-come, first-served* system where early withdrawers get their full amount while late withdrawers cannot exit fully until the deficit is filled. This breaks the expected *all shares are equally backed* invariant and creates an unfair burden on lenders who withdraw last.

Example:

1. Lender A deposits 500 USDC and receives 500 shares, Lender B deposits 500 USDC and receives 500 shares
2. Borrower takes out a loan of 900 USDC, strategy underperforms, returns 700 USDC
3. Global deficit: 200 USDC
4. Lender A withdraws first: Gets 500 USDC (full amount)
5. Lender B withdraws second: Gets 300 USDC (limited by vault balance)
6. Lender B has 200 shares remaining until deficit filled

This issue is classified as **Low** severity because the last lender cannot exit until the deficit is covered, creating unfair distribution among existing participants and making new depositors unsecured lenders of last resort.

Recommendation

We recommend accounting for the current deficit in the getMaxWithdraw() function. This can be achieved by calculating the deficit per share value and then subtracting the shares that cover the deficit from what the lender can actually withdraw. Once the deficit is covered, the lender will be able to withdraw the rest of their shares.

Client's Commentary

We accept the risk of the uneven withdrawals (where the first user to withdraw can take everything available, and the last user only gets the residual balance) in favor of UX and operational guarantees.Introducing a deficit-checking mechanism would create a situation where, even when liquidity is present, users would be unable to withdraw 100% of their funds in a single transaction. All users would have to wait for the deficit to be replenished and then perform an additional dust withdrawal, which is rather inconvenient and increases gas costs. The protocol takes on the operational obligation to promptly cover any emerging deficit through the fillBaseAssetDeficit function. This makes any impermanent losses a temporary technical condition rather than a systemic problem. We expect that users will not have a rational incentive to trigger a bank run, since they are guaranteed that any potential loss is only impermanent in nature and will be fully covered by the protocol. In other words, the current implementation is optimized for the happy path, where any deficits are an unlikely temporary exception eventually resolved by the administrator, rather than shifted onto users through complicated haircut mechanisms.

L-2	Inaccurate NatSpec Documentation for liquidate() Function		
Severity	Low	**Status**	Fixed in 0350a678

Description

The NatSpec documentation for the LiquidationFacet.liquidate() function incorrectly describes the positionData parameter structure. The documentation states:

```
* @param positionData The vaults and positions to liquidate and min amount
* of assets to receive and bytes32 of "data" array has packed vault address
* and position id min amount of assets to receive and some platform-dependent
* data like [vaultAddress1, positionId1, data1, minAmount1,
* vaultAddress2, positionId2, data2, minAmount2, ...]
* for Pendle data is ytIndex
```

However, the actual implementation expects vaultAddress1andPositionId1 to be a single packed variable, not separate vaultAddress1 and positionId1 variables as suggested by the documentation.
This documentation mismatch could lead to confusion for developers trying to understand the function's expected input format.

Recommendation

We recommend updating the NatSpec documentation to accurately reflect the actual parameter structure, clarifying that vault address and position ID are packed into a single variable rather than being separate array elements.

Client's Commentary

✓ You are absolutely right, has recently fixed PR-31 in the PR

L-3	Unnecessary End Date Check in Strategy Loop		
Severity	Low	**Status**	Fixed in 0350a678

Description

The `BaseLogicLib.updateLenderPointsForAllStrategies()` function contains an unnecessary check for strategy end date. The condition:

```
if (
BaseStorageLib.getEndDate(i) < block.timestamp
    && BorrowLogicLib.allBorrowersAreFinished(i)
)
```

The first part `BaseStorageLib.getEndDate(i) < block.timestamp` is unnecessary because `allBorrowersAreFinished(i)` already includes this exact check internally:

```
function allBorrowersAreFinished(
    uint256 strategyId
) internal view returns (bool) {
    return BaseStorageLib.getEndDate(strategyId) < block.timestamp
        && BorrowStorageLib
            .getStrategyActiveBorrowerPositionCount(strategyId) == 0;
}
```

The same issue exists in the `BaseStorageLib.withdrawYield()` function:

```
require(
    BaseStorageLib.getEndDate(strategyId) < block.timestamp
        && BorrowLogicLib.allBorrowersAreFinished(strategyId),
    ErrorLib.BorrowersNotFinished(strategyId)
);
```

These duplicate checks waste gas and make the code less maintainable.

Recommendation

We recommend removing the unnecessary `BaseStorageLib.getEndDate(strategyId) < block.timestamp` checks in both functions.

Client's Commentary

✓ PR-31! Thank you

L-4	Inconsistent Comparison Operator in Strategy End Date Check		
Severity	Low	**Status**	Fixed in 0350a678

Description

The `BaseLogicLib.updateLenderPoints()` function uses an inconsistent comparison operator when checking strategy end dates. The function uses `>=` while the similar logic in `getAccruedLenderYieldOneStrategy()` uses `>`.

In `BaseLogicLib.updateLenderPoints()`:

```
if (strategyEndDate >= lastUpdateTime) {
```

In `BaseLogicLib.getAccruedLenderYieldOneStrategy()`:

```
if (strategyEndDate > lastUpdate) {
```

When `strategyEndDate == lastUpdateTime`, the `>=` operator causes the function to enter the if block and calculate `timeDelta = 0` in both branches (whether `block.timestamp > strategyEndDate` or not). This results in unnecessary zero-duration yield calculations and wasted gas, while the `>` operator would skip this processing entirely.

Recommendation

We recommend standardizing the comparison operator to use `>` in both functions for consistency, as this avoids processing zero-duration intervals and matches the intended logic of only calculating yield when there's actual time elapsed.

Client's Commentary

✓ *Fixed PR-31*

L-5	Unused Function setBorrowApr() in BaseStorageLib		
Severity	Low	**Status**	Fixed in 0350a678

Description

The BaseStorageLib.setBorrowApr() function is defined but never called anywhere in the codebase. This function allows setting the borrow APR for a strategy but remains unused, creating dead code that increases contract size and gas deployment costs.

```
function setBorrowApr(uint256 strategyId, uint256 apr) internal {
    ensureValidStrategyId(strategyId);
    require(apr <= MAX_APR, ErrorLib.InvalidBorrowApr());
    getStorage().strategies[strategyId].borrowApr = uint40(apr);
    emit EventLib.SetBorrowApr(strategyId, apr);
}
```

The function includes proper validation and emits events, suggesting it was intended for use but was never integrated into the protocol's functionality.

Recommendation

We recommend either removing the unused setBorrowApr() function to reduce contract size and deployment costs, or implementing it in the appropriate location if dynamic APR updates are intended functionality.

Client's Commentary

✓ Nice find, PR-31 the function as we don't change APR for existing strategies

L-6	MIN_COLLATERAL_GIVEN Check Inconsistent Across Token Decimals		
Severity	Low	**Status**	Fixed in 0350a678

Description

The MIN_COLLATERAL_GIVEN constant is set to 1e6 (1 USDC/USDT) but is used as a universal minimum collateral check for all token types. This creates inconsistent behavior across different token decimals:

- 6-decimal tokens (USDC/USDT): 1e6 = 1 token (reasonable minimum)
- 18-decimal tokens (DAI/ETH): 1e6 = 0.000000000001 token (negligible dust amount)
- 8-decimal tokens (WBTC): 1e6 = 0.01 token (potentially too high, excluding small positions)

```
require(
    collateralAmount >= MIN_COLLATERAL_GIVEN,
    ErrorLib.NotEnoughCollateral(collateralAmount, MIN_COLLATERAL_GIVEN)
);
```

Recommendation

We recommend implementing token-specific minimum collateral checks based on the token's decimal precision.

Client's Commentary

1. ✓ PR-31 for base asset. 2) ✓ PR-31 for PT-asset (strategy asset)

L-7	delegatecall Return Value Not Checked in Adapter Emergency Function		
Severity	Low	**Status**	Fixed in 0350a678

Description

The Adapter.delegatecall() function performs a delegatecall to an arbitrary target but does not verify that the call succeeded.

```
function delegatecall(
    address target,
    bytes memory data
) external onlyOwner returns (bool, bytes memory) {
    return target.delegatecall(data);
}
```

Recommendation

We recommend checking the return value and reverting on failure to ensure callers are immediately aware of unsuccessful recovery attempts.

Client's Commentary

✓ Fixed PR-31 and PR-31 (forgot to amend)

L-8	Vault Removal Breaks Liquidation System		
Severity	Low	**Status**	Fixed in 0350a678

Description

The `VaultRegistry.removeVault()` function allows the curator to remove vaults from the registry, but this breaks the liquidation system for any active borrower positions in those vaults. When a vault is removed via `removeVault()`, the `prepareLiquidations()` function will revert with `VaultNotFound` error when trying to liquidate positions in the removed vault, making it impossible to liquidate undercollateralized positions.

Recommendation

We recommend preventing removal of vaults that have active borrower positions by adding a check in `removeVault()` to ensure no active positions exist, or consider removing the `removeVault()` function entirely if it serves no essential purpose in the system.

Client's Commentary

✓ *Fixed PR-31*

L-9	Hardcoded Curve Pool Token Indexes Limit Adapter Flexibility		
Severity	Low	**Status**	Fixed in 0350a678

Description

The `CurveStableSwapAdapter` hardcodes token indexes `0` and `1` in when calling `ICurveStableSwap.exchange()`. This design assumes `TOKEN_IN` is always at index `0` and `TOKEN_OUT` at index `1` within the Curve pool.

While this works correctly for pools deployed with this specific token ordering, it reduces the adapter's reusability. For pools where tokens are positioned in reverse order, a separate adapter instance must be deployed. This increases deployment overhead and requires careful verification during strategy setup to ensure the pool's token order matches the adapter's expectations.

Recommendation

We recommend making token indexes configurable via constructor immutable parameters to improve adapter flexibility.

Client's Commentary

✓ *PR-31, made more flexible*

L-10	Race Condition: minAssetsOut Denomination May Mismatch After Route Update		
Severity	Low	Status	Fixed in ddcd3f50

Description

A user signs a transaction with minAssetsOut[] aligned to the currently published route. If a curator updates the route before inclusion and an intermediate hop now operates with different assets (or their order), the numeric thresholds minAssetsOut[i] are applied to different tokens than the user intended. This denomination mismatch can make slippage checks pass erroneously and lead to unexpected execution outcomes.

The problem is further enabled by asynchronous adapters: when an adapter returns finished = false, the final check that the ultimate assetOut matches the vault's base asset is not executed in that call.

Recommendation

We recommend including a route hash or version in the transaction data and revert if the on-chain route at execution time differs from the provided hash/version.

Client's Commentary

***Client**: Fixed PR-42: swaps revert if the strategy `route` / `reverseRoute` was updated within the last 15 minutes*

***MixBytes**: This issue has been marked as fixed. As a side effect, liquidation operations may revert during the cooldown period (unborrow operations are also affected). The client acknowledges this availability trade-off and accepts the associated risk. This risk is considered low-probability, as it requires a route update to occur shortly before positions become liquidatable during volatile market conditions.*

L-11	Curve Adapter Accepts Identical Token Indexes		
Severity	Low	**Status**	Fixed in ddcd3f50

Description

`CurveStableSwapAdapter` does not validate that `INPUT_TOKEN_INDEX` and `OUTPUT_TOKEN_INDEX` are different. A misconfiguration can result in `exchange(i, i, amount, minAmountOut)`, which typically reverts in Curve pools. Given that indexes are immutable constructor parameters set by trusted admins, we classify the issue as Low severity.

Recommendation

We recommend adding a constructor check to ensure `_inputTokenIndex != _outputTokenIndex`.

Client's Commentary

Fixed PR-42

L-12	Permissive Default Whitelist in Adapters Allows Any Caller		
Severity	Low	**Status**	Fixed in ddcd3f50

Description

The base `Adapter` provides a default whitelist implementation `_isVaultWhitelisted(address)` that returns `true`. As a result, `buy()` and `continueSwap()` become callable by any address in adapters that do not override this function, expanding the callable surface beyond trusted vaults. While current adapters are stateless and typically require the caller's tokens, this permissive default is risky for future implementations (especially around asynchronous continuations) and undermines the intent to restrict adapter execution to specific vaults.

Recommendation

We recommend removing the permissive default by making `_isVaultWhitelisted()` abstract so each adapter must explicitly declare allowed vaults at compile time.

Client's Commentary

***Client**: Acknowledged. A vault whitelist is not required for all adapters: many are synchronous/stateless "pure swap" wrappers where arbitrary callers can only act on their own funds. We keep whitelisting as an opt-in via overrides (e.g., AdapterWithWhitelist) and will enforce it for adapters that are permissioned/stateful or involve async continuations.*

***MixBytes**: We've marked the issue as fixed, as an `AdapterWithWhitelist` contract has been introduced.*

L-13	Liquidation Request Event Includes Skipped Positions		
Severity	Low	**Status**	Fixed in ddcd3f50

Description

In BorrowLogicLib.liquidationRequest(), the function skips positions that have unfinishedSwap or are not in activeBorrowerPositionIds, but the event is emitted inside the loop on each iteration with the entire input array, instead of being emitted once after the loop with only the positions that were actually marked for liquidation.

```
for (uint256 i = 0; i < positionsLength; i++) {
    uint256 positionId = positions[i];

    if (
        BorrowStorageLib.hasUnfinishedSwap(positionId)
        || !$.activeBorrowerPositionIds.contains(positionId)
    ) {
        continue; // Position is skipped
    }

    BorrowStorageLib.setSubjectToLiquidation(positionId);
    // Emits entire array on each iteration
    IVaultRegistry(...).emitLiquidationRequest(positions);
}
```

This results in:

1. The event being emitted multiple times (once per valid position) instead of once with all processed positions
2. Each event containing the entire input array, including skipped positions, rather than only the positions that were actually marked for liquidation

For example, if positions [1, 2, 3] are passed but only position 2 is valid, the event will be emitted once with array [1, 2, 3], suggesting all three positions were processed when only position 2 was actually marked. This can confuse off-chain monitoring systems.

This issue is classified as **Low** severity because it doesn't affect protocol security or functionality, but it represents a data inconsistency that can mislead off-chain systems relying on events for tracking liquidation requests.

Recommendation

We recommend collecting valid position IDs during the loop and emitting the event once after the loop completes, with an array containing only the positions that were actually marked for liquidation.

Client's Commentary

Fixed PR-41

L-14	Unreachable Code in Adapter.continueSwap()		
Severity	Low	**Status**	Fixed in ddcd3f50

Description

In Adapter.continueSwap(), two code branches are unreachable due to earlier validation:

1. The check if (amountOut > 0) is always true because the function enforces require(tokenOutBalanceAfter > tokenOutBalanceBefore, ErrorLib.InvalidAmountOut()) before calculating amountOut, ensuring it is strictly greater than zero for any successful execution.

2. The check if (finished && amountOut == 0) is unreachable because if amountOut == 0, the function would have already reverted on the earlier validation that requires tokenOutBalanceAfter > tokenOutBalanceBefore.

Recommendation

We recommend removing the redundant if (amountOut > 0) and the unreachable if (finished && amountOut == 0) checks.

Client's Commentary

Fixed, mostly by PR-41 changes

3. About MixBytes

MixBytes is a leading provider of smart contract audit and research services, helping blockchain projects enhance security and reliability. Since its inception, MixBytes has been committed to safeguarding the Web3 ecosystem by delivering rigorous security assessments and cutting-edge research tailored to DeFi projects.

Our team comprises highly skilled engineers, security experts, and blockchain researchers with deep expertise in formal verification, smart contract auditing, and protocol research. With proven experience in Web3, MixBytes combines in-depth technical knowledge with a proactive security-first approach.

Why MixBytes

- **Proven Track Record:** Trusted by top-tier blockchain projects like Lido, Aave, Curve, and others, MixBytes has successfully audited and secured billions in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems.
- **Innovative Research:** Our team actively contributes to blockchain security research, sharing knowledge with the community.

Our Services

- **Smart Contract Audits:** A meticulous security assessment of DeFi protocols to prevent vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical research and security modeling for Web3 projects.
- **Custom Security Solutions:** Tailored security frameworks for complex decentralized applications and blockchain ecosystems.

MixBytes is dedicated to securing the future of blockchain technology by delivering unparalleled security expertise and research-driven solutions. Whether you are launching a DeFi protocol or developing an innovative dApp, we are your trusted security partner.

Contact Information


https://mixbytes.io/


https://github.com/mixbytes/audits_public


hello@mixbytes.io


https://x.com/mixbytes