Security Assessment

ether.fi – Cash Module Combined Audit Report

April-June 2025

Prepared for ether.fi

certora

ether.fi

Table of contents

Project Summary

Project Scope

Project Name	Initial Commit Hash	Latest Commit Hash	Platform	Start Date	End Date
Liquid Minting and Staking module	Hash	Hash	EVM	01/04/2025	03/04/2025
Default module changes	Hash	Hash	EVM	03/04/2025	05/04/2025
TopUp indexing with TxID	Hash	Hash	EVM	07/04/2025	09/04/2025
Multi spend and referrals	Hash	Hash	EVM	09/04/2025	11/04/2025
Weth Topups	Hash	Hash	EVM	11/04/2025	12/04/2025
Multiple Settlement Dispatchers	Hash	Hash	EVM	14/04/2025	15/04/2025
Stargate Bridge Module	Hash	Hash	EVM	15/04/2025	17/04/2025
Aave Module	Hash	Hash	EVM	17/04/2025	19/04/2025
NTT Bridge Adapter	Hash	Hash	EVM	21/04/2025	22/04/2025

OpenOcean Swap Fix	Hash	Hash	EVM	02/05/2025	06/05/2025
Cash Module Additional Audit Round	Hash	Hash	EVM	02/05/2025	08/05/2025
Newly uncovered findings	Hash	Hash	EVM	04/05/2025	18/05/2025
Liquid withdrawals, refund wallet and OpenOcean fix	Hash	Hash	EVM	22/05/2025	27/05/2025
CanSpend() audit	Hash	Hash	EVM	04/06/2025	05/06/2025
Improved max spend functions on Cash	Hash	Hash	EVM	11/06/2025	13/06/2025
Scroll ERC20 Bridge Adapter	Hash	Hash	EVM	16/06/2025	18/06/2025
Cashback upgrade	Hash	Hash	EVM	16/06/2025	18/06/2025

Project Overview

This document describes the manual code review of several modules and changes to the cash-v3 repository.

The work was about a 47 day effort undertaken from **01/04/2025** to **18/06/2025**

The team performed a manual audit of all the Solidity smart contracts. During the manual audit, the Certora team discovered bugs in the Solidity smart contracts code, as listed on the following page.

The latest commit hash which was reviewed is [8d7bd4b]

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	1	1	1
Medium	10	10	9
Low	12	12	8
Informational	24	24	20
Total	47	47	39

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
Liquid Minting and Staking module			
-	-	-	-
Default module changes			
-	-	-	-
TopUp indexing with TxID			
L-01	Potential Race Condition in markCompletedAdmin Flow Can Lead to Double Top-Ups	Medium	Acknowledged
Multi spend and referrals			
L-01	referrer can be the same as the safe or the spender	Medium	Acknowledged
Weth Topups			
-	-	-	-
Multiple Settlement Dispatchers			
-	-	-	-

Stargate Bridge Module			
M-01	StargateModule::_bridgeOFT might leave unused approval which causes problems with USDT	Medium	Fixed
Aave Module			
M-01	AaveV3Module::_repay might leave unused approval which causes problems with USDT	Medium	Fixed
M-02	Withdrawing ETH from Aave with amount = uint256.max will revert	Medium	Fixed
M-03	Missing functionality to claim Aave incentive rewards	Medium	Fixed
NTT Bridge Adapter			
-	-	-	-
OpenOcean Swap Fix			
-	-	-	-
Liquid withdrawals, refund wallet and OpenOcean fix			
M-01	Missing slippage protection on liquid withdrawals	Medium	Fixed
L-01	Liquid withdrawal path limited to a single preconfigured asset	Medium	Fixed

CanSpend() audit			
-	-	-	-
Cash Module Additional Audit Round			
H-01	Improper debt conversion during liquidation	High	Fixed
M-01	Improper liquidation bonus calculation	Medium	Fixed
M-02	Debt supplier funds can get locked in the manager	Medium	Fixed
M-03	No check for duplicate tokens in canSpend causes improper debt calculation	Medium	Fixed
L-01	Token deduplication not implemented upon spending	Medium	Fixed
L-02	Callback from cashback token to retrieve pending cashback can drain cashback tokens	Medium	Fixed
L-03	Normalized amount calculation should always round up	Medium	Fixed
L-04	Max can spend returns 0 if withdrawal exists for a token	Medium	Fixed

Newly Introduced Issues			
M-01	Malicious actor can grief interest index accrual	Medium	Acknowledged
M-02	Sending max amounts will not work in the AAVE module for ETH	Medium	Fixed
L-01	Malicious actor can grief the TopUp Factory by causing more fees to be paid than necessary	Low	Fixed
L-02	ETH is not recoverable	Low	Fixed
L-03	Anyone can prevent borrow tokens from getting disabled	Low	Acknowledged
L-04	Oracle prices for stable tokens are prone to value extraction	Low	Acknowledged
Improved max spend functions on Cash			
-	-	-	-
Scroll ERC20 Bridge Adapter			
-	-	-	-
Cashback upgrade			
L-01	Inconsistent update of totalCashbackEarnedInUsd state variable	Low	Fixed

Liquid Minting and Staking module

Project Overview

This report presents the findings of a manual code review for the **Liquid Minting and Staking module** audit within the **EtherFi Cash** project. The work was undertaken from **April 1st to April 2nd 2025**

The following contract list is included in the scope of this audit:

- src/modules/etherfi/EtherFiStakeModule.sol
- src/modules/etherfi/EtherFiLiquidModule.sol
- src/top-up/bridge/EtherFiLiquidBridgeAdapter.sol

The code modifications examined during this review were implemented in the following pull request - PR#11

Informational Issues

I-01. EtherFiStakeModule doesn't check for address(0) in the constructor

Description: The constructor of EtherFiStakeModule does not validate whether the provided addresses (_dataProvider, _syncPool, _weth, and _weETH) are non-zero. This omission contrasts with other contracts in the codebase that implement such checks.

Recommendation: Add address(0) checks in the constructor

Customer's response: Fixed in commit fee1fde

Fix Review: Fix confirmed

I-02. Misleading EIP-712 Compatibility Comment in `_getDepositDigestHash`

Description: The function EtherFiStakeModule::_getDepositDigestHash claims to return an "EIP-712 compatible digest hash for signature verification," but this is incorrect. The function applies toEthSignedMessageHash(), which follows the Ethereum Signed Message (EIP-191) format, not EIP-712.

Similarly, EtherFiLiquidModule::_getDepositDigestHash has the same issue.

Recommendation: Update the comments to correctly state that the functions follow EIP-191 instead of EIP-712.

Customer's response: Fixed in commits fee1fde and 77260a7

Fix Review: Fix confirmed

I-03. Missing Vault Consistency Check in `EtherFiLiquidModule` Constructor

Description: In EtherFiLiquidModule, the addLiquidAssets function ensures that each teller's vault matches the corresponding liquid asset. However, the constructor does not perform this check when initializing _assets and _tellers. This inconsistency could allow an invalid configuration where a teller's vault does not correspond to the assigned liquid asset.

Recommendation: Add the missing vault validation in the constructor to ensure that the assigned tellers correctly map to their respective liquid assets at initialization.

Customer's response: Fixed in commit fee1fde

Fix Review: Fix confirmed

Default module changes

Project Overview

This report presents the findings of a manual code review for the **Default module changes** audit within the **EtherFi Cash** project. The work was undertaken from **April 3rd to April 4th 2025**

The following contract list is included in the scope of this audit:

- src/data-provider/EtherFiDataProvider.sol
- src/interfaces/IEtherFiDataProvider.sol
- src/safe/EtherFiSafe.sol
- src/safe/ModuleManager.sol

The code modifications examined during this review were implemented in the following pull request - PR#12

TopUp indexing with TxID

Project Overview

This report presents the findings of a manual code review for the **TopUp indexing with TxID** audit within the **EtherFi Cash** project. The work was undertaken from **April 7th to April 8th 2025**

The following contract list is included in the scope of this audit:

- src/top-up/TopUpDest.sol

The code modifications examined during this review were implemented in the following pull request - PR#14

Low Severity Issues

L-01 Potential Race Condition in markCompletedAdmin Flow Can Lead to Double Top-Ups

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: UpgradeTopUpTxId.s.sol	Status: Acknowledged	

Description: If a TopUp transaction is triggered between the call to upgradeToAndCall and the actual execution of markCompletedAdmin, then _topUp() will proceed with the transfer because the transaction is not yet marked as completed. This results in double-spending — once during the previous _topUp and now.

Since Foundry does not guarantee strict sequential execution in practice, this could happen unintentionally during script execution.

Recommendations: Perform the markCompletedAdmin logic within the upgradeToAndCall call by encoding it as a function call. This ensures the transactions are marked as completed atomically during the upgrade, leaving no gap for unexpected calls to _topUp

Customer's response: We are fine with the finding since we will disable topups while we upgrade

Informational Severity Issues

I-01. Missing NatSpec Documentation in TopUpDestWithMarkCompleteAdmin::markCompletedAdmin

Description: The markCompletedAdmin function in TopUpDestWithMarkCompleteAdmin lacks NatSpec documentation

Recommendation: Add appropriate NatSpec comments to the markCompletedAdmin function

Customer's response: Since we are discarding the TopUp with markCompletedAdmin function, it is fine

I-02. markCompletedAdmin Missing Length Validation for chainIds and amounts Arrays

Description: The markCompletedAdmin function in TopUpDestWithMarkCompleteAdmin performs a batch update to mark multiple transactions as completed. While it validates that txHashes, users, and tokens arrays are of equal length, it does not validate the lengths of chainIds and amounts.

Recommendation: Add explicit checks to ensure that chainIds.length == len and amounts.length == len alongside the existing validations.

Customer's response: Since we are discarding the TopUp with markCompletedAdmin function, it is fine

Multi spend and referrals

Project Overview

This report presents the findings of a manual code review for the **Multi spend and referrals** audit within the **EtherFi Cash** project. The work was undertaken from **April 9th to April 10th 2025**

The following contract list is included in the scope of this audit:

- src/modules/cash/CashModuleStorageContract.sol
- src/modules/cash/CashModuleSetters.sol
- src/modules/cash/CashModuleCore.sol
- src/modules/cash/CashLens.sol
- src/modules/cash/CashEventEmitter.sol

The code modifications examined during this review were implemented in the following pull request - PR#15

Low Severity Issues

L-01 referrer can be the same as the safe or the spender

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: CashModuleCore.sol	Status: Acknowledged	

Description: In the spend() function, while it is correctly checked that spender != safe, there is no validation for the referrer. This allows the referrer to be the same as the safe or the spender, which is unintended. If the referrer is the same as either of them, it effectively means the safe/spender receives the referral bonus themselves — which should not be allowed.

Recommendations: Check if referrer is not the same as safe and spender

Customer's response: We don't want to add condition on chain for this, we would implement an off-chain solution for now

Weth Topups

Project Overview

This report presents the findings of a manual code review for the **Weth Topups** audit within the **EtherFi Cash** project. The work was undertaken from **April 11th to April 11th 2025**

The following contract list is included in the scope of this audit:

- src/top-up/bridge/StargateAdapter.sol
- src/top-up/TopUpDestNativeGateway.sol

The code modifications examined during this review were implemented in the following commit - 6b17e7f

Multiple Settlement Dispatchers

Project Overview

This report presents the findings of a manual code review for the **Multiple Settlement Dispatchers** audit within the **EtherFi Cash** project. The work was undertaken from **April 14th to April 14th 2025**

The following contract list is included in the scope of this audit:

- src/debt-manager/DebtManagerCore.sol
- src/modules/cash/CashEventEmitter.sol
- src/modules/cash/CashModuleCore.sol
- src/modules/cash/CashModuleSetters.sol
- src/modules/cash/CashModuleStorageContract.sol
- src/settlement-dispatcher/SettlementDispatcher.sol
- src/interfaces/IDebtManager.sol
- src/interfaces/ICashModule.sol
- src/interfaces/ICashEventEmitter.sol

The code modifications examined during this review were implemented in the following pull request - PR#16

Stargate Bridge Module

Project Overview

This report presents the findings of a manual code review for the **Stargate Bridge Module** audit within the **EtherFi Cash** project. The work was undertaken from **April 15th to April 16th 2025**

The following contract list is included in the scope of this audit:

- src/modules/stargate/StargateModule.sol

The code modifications examined during this review were implemented in the following commit hash - 2b37e64

Medium Severity Issues

M-01 StargateModule::_bridgeOFT might leave unused approval which causes problems with USDT

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: StargateModule.sol	Status: Fixed	

Description: In the StargateModule::_bridgeOft function, if the OFT requires approval, the full amount is approved for the OFT contract to spend. However LayerZero may apply dust removal logic, which means the actual amount spent can be slightly lower than the approved amount. This can leave a non-zero, unused allowance behind.

This behavior is especially problematic for tokens like USDT, which enforce strict approval mechanics: if a non-zero allowance already exists, a new approval call with a different amount will revert unless the allowance is first set to zero.

Recommendations: Approve to 0 value after bridging

Customer's response: Fixed in commit 16143c1

Fix Review: Fix confirmed

Aave Module

Project Overview

This report presents the findings of a manual code review for the **Aave Module** audit within the **EtherFi Cash** project. The work was undertaken from **April 17rd to April 18th 2025**

The following contract list is included in the scope of this audit:

- src/modules/aave-v3/AaveV3Module.sol

The code modifications examined during this review were implemented in the following commit hash - 8364c6d

Medium Severity Issues

M-01 AaveV3Module::_repay might leave unused approval which causes problems with USDT

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AaveV3Module.sol	Status: Fixed	

Description: In the AaveV3Module::_repay function, if the specified amount is greater than the actual debt owed to Aave, the protocol will leave unused approval as Aave will only pull the owed amount. This is particularly problematic for tokens like USDT, which revert on subsequent approve calls unless the current allowance is first set to zero. As a result, future repay calls involving USDT will fail due to this stale allowance.

Recommendations: Approve to 0 value after repaying

Customer's response: Fixed in commit 23e2a6d

Fix Review: Fix confirmed

M-02 Withdrawing ETH from Aave with amount = uint256.max will revert

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AaveV3Module.sol	Status: Fixed	

Description: In the AaveV3Module::_withdraw function, when withdrawing ETH from Aave, the contract first withdraws WETH from the Aave pool and then calls WETH.withdraw(amount) to unwrap it. However, it is common practice to use amount = type(uint256).max to withdraw the user's full balance. While Aave handles uint256.max by returning the full balance internally, the subsequent WETH.withdraw(uint256.max) will revert, since the safe obviously doesn't hold that amount of WETH — it only holds the actual withdrawn balance.

Recommendations: Either delegate the entire ETH withdrawal logic to WrappedTokenGatewayV3::withdrawETH, which already handles uint256.max correctly, or replicate its logic

Customer's response: Fixed in commit 4ca0132

Fix Review: Fix confirmed

certora

ether.fi

M-03 Missing functionality to claim Aave incentive rewards		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AaveV3Module.sol	Status: Fixed	

Description: The AaveV3Module currently lacks functionality to claim Aave incentive rewards that safes accumulate over time through Aave's rewards program. When safes supply or borrow assets on specific networks and tokens, Aave distributes incentive tokens (e.g., AAVE, stkAAVE) via its RewardsController. These tokens accumulate but remain unclaimed unless explicitly withdrawn through the claimRewards() function.

Without a mechanism to claim them, the rewards remain stuck in Aave's controller, resulting in a loss of value for the safes that interact with the AaveV3Module.

Aave docs - https://aave.com/docs/primitives/incentives

Current active incentive programs - https://apps.aavechan.com/merit

Recommendations: Consider adding a separate function to claim Rewards from Aave RewardsController

Customer's response: Fixed in commit c01f45b

Fix Review: Fix confirmed

NTT Bridge Adapter

Project Overview

This report presents the findings of a manual code review for the **NTT Bridge Adapter** audit within the **EtherFi Cash** project. The work was undertaken from **April 21st to April 21st 2025**

The following contract list is included in the scope of this audit:

- src/top-up/bridge/NTTAdapter.sol
- src/interfaces/INTTManager.sol

The code modifications examined during this review were implemented in the following pull request - PR#18

OpenOcean Swap Fix

Project Overview

This report presents the findings of a manual code review for the **OpenOcean Swap Fix** audit within the **EtherFi Cash** project. The work was undertaken from **May 2nd to May 6th 2025**

The following contract list is included in the scope of this audit:

- src/modules/openocean-swap/OpenOceanSwapModule.sol

The code modifications examined during this review were implemented in the following pull request - PR#22

Informational Severity Issues

I-01. Unnecessary reset of token allowance to zero after swap

Description:
https://github.com/etherfi-protocol/cash-v3/pull/22/files#diff-500eb63834b20c8867e66dc30c3241a7d0908a8aa4e6f0417a33234214e3457dR219

In the OpenOceanSwapModule::_swapERC20 function, after approving fromAssetAmount to the swap router and executing the swap, the allowance is explicitly reset to zero. While this is a general best practice to prevent leftover approvals, in this specific case, the router being used (OpenOcean) always utilizes the exact approved amount and fully spends the allowance during the swap. As a result, there will be no residual approval left.

This makes the final approval to zero redundant.

Recommendation: Remove the call that approves to 0 value

Customer's response: Acknowledged

I-02. Missing validation for flags field in OpenOcean swap description

Description:
https://github.com/etherfi-protocol/cash-v3/blob/fix/swaps/src/modules/openocean-swap/OpenOceanSwapModule.sol#L269-L274
In the OpenOceanSwapModule::_validateSwapData function, key fields of the OpenOceanSwapDescription struct are validated to ensure correctness and integrity of the swap parameters. However, the flags field is not currently checked. This field is relevant for specifying the nature of the asset being swapped (e.g., native or ERC20), and incorrect values may lead to unexpected behavior during swap execution.

Recommendation: Validate the flags field to equal 0 when native swaps and 2 when doing ERC20 swaps

Customer's response: Acknowledged

Liquid withdrawals, refund wallet and OpenOcean fix

Project Overview

This report presents the findings of a manual code review for the **Liquid withdrawals, refund wallet and OpenOcean fix** audit within the **EtherFi Cash** project. The work was undertaken from **May 22nd to May 27th 2025**

The following contract list is included in the scope of this audit:

- src/data-provider/EtherFiDataProvider.sol
- src/interfaces/IBoringOnChainQueue.sol
- src/interfaces/IEtherFiDataProvider.sol
- src/modules/etherfi/EtherFiLiquidModule.sol
- src/modules/openocean-swap/OpenOceanSwapModule.sol
- src/settlement-dispatcher/SettlementDispatcher.sol

The code modifications examined during this review were implemented in the following pull request - PR#27

Medium Severity Issues

M-01 Missing slippage protection on liquid withdrawals

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: EtherFiLiquidModule.sol	Status: Fixed	

Description: The withdraw() function in EtherFiLiquidModule lacks slippage protection. Since withdrawals go through BoringOnChainQueue, which converts liquid asset shares into an assetOut amount based on a dynamic price, users may receive significantly less than expected if prices shift before execution.

```
277 +     function withdraw(address safe, address liquidAsset, uint256 amountToWithdraw, address signer, bytes calldata signature) external
    onlyEtherFiSafe(safe) onlySafeAdmin(safe, signer) {
278 +         bytes32 digestHash = _getWithdrawDigestHash(safe, liquidAsset, amountToWithdraw);
279 +         _verifyAdminSig(digestHash, signer, signature);
280 +         _withdraw(safe, liquidAsset, amountToWithdraw);
281 +     }
282 +
```

Recommendations: Add slippage protection to the EtherFiLiquidModule::withdraw(). You can do that by calling BoringOnChainQueue::previewAssetsOut() and comparing the returned amount with a user-specified one.

Customer's response: Fixed in commit 0387139

Fix Review: Fix confirmed

Low Severity Issues

L-01 Liquid withdrawal path limited to a single preconfigured asset

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: EtherFiLiquidModule.sol	Status: Fixed	

Description: The EtherFiLiquidModule::withdraw() function only supports withdrawing a single preconfigured underlying asset (assetOut) per liquid token. This restriction comes from the static liquidWithdrawConfig[liquidAsset].assetOut, which is not user-selectable at call time.

In situations where the configured assetOut hits its withdrawCapacity, users are blocked from withdrawing, even if other underlying assets (also supported by the liquid asset) still have sufficient capacity. This creates unnecessary friction and introduces centralization risk, since users must wait for an admin to update the configuration.

Recommendations: Allow users to specify assetOut dynamically at withdrawal time, rather than relying on a hardcoded config

Customer's response: Fixed in commit 8359db8

Fix Review: Fix confirmed

Informational Severity Issues

I-01. Incorrect comment in setLiquidAssetWithdrawConfig

https://github.com/etherfi-protocol/cash-v3/pull/27/commits/303ea61d0728be121cc51e976e3ed83480d1467b#diff-3ac53c66e982cdfdd836113f5a53fc19ccb9702e9a475661536066d28103d846R471

https://github.com/etherfi-protocol/cash-v3/pull/27/commits/125083d7c936474c7f524c8d814ab07805c58fb6#diff-77edbb8a23b7d6dde306c34ff4afe1de1cd694ddfc1bf56b70aceae3e2dfca69R272

The function setLiquidAssetWithdrawConfig claims the discount parameter uses 5 decimals of precision (1% = 100_000), but it is declared as a uint16, which can only store values up to 65,535.

Recommendation: Update the comment

Customer's response: Fixed in commit 6c8d7ec

Fix Review: Fix confirmed

Can Spend

Project Overview

This report presents the findings of a manual code review for the **Can Spend** audit within the **EtherFi Cash** project. The work was undertaken from **June 3rd to June 4th 2025**

The following contract list is included in the scope of this audit:

- src/modules/cash/CashLens.sol

The code modifications examined during this review were implemented in the following pull request - PR#30

Informational Severity Issues

I-01. CanSpend() view function is inconsistent with the logic of the state changing spend()

Description: Calling CanSpend() on CashLens.sol is a view only function that simulates the execution of spend() in CashModule to calculate if an account can spend a particular amount of tokens.

The problem is that CanSpend() internally calls _validateSpending() which evaluates the spending mode like this:

```
JavaScript
if (safeData.incomingCreditModeStartTime != 0)
            safeData.mode = Mode.Credit;
```

The same applies for the newly introduced canSpendSingleToken():

```
JavaScript
function canSpendSingleToken(...) public view returns (Mode mode, address token, bool
canSpendResult, string memory declineReason) {
        ....
        if (safeData.incomingCreditModeStartTime != 0) safeData.mode = Mode.Credit;
        mode = safeData.mode;
       ....
}
```

While the state changing spend() function works like this:

```
JavaScript
if (
            $.incomingCreditModeStartTime != 0 &&
            block.timestamp > $.incomingCreditModeStartTime
        ) {
            $.mode = Mode.Credit;
            delete $.incomingCreditModeStartTime;
        }
```

As you can see, the actual state changing function spend() changes the mode to credit only after the incomingCreditModeStartTime has passed.

CanSpend() on the other hand switches to Credit mode only by checking if incomingCreditModeStartTime has been set, but without validating if start time has come.

Impact: The _validateSpending() & canSpendSingleToken() functions would sometimes improperly evaluate spending under Credit mode and return wrong data, which would be inconsistent with the actual behaviour of spend()

Recommendations: Make sure to also check that startTime has arrived before switching to Credit in _validateSpending() & canSpendSingleToken()

Customer's response: It is an intentional condition. The reason for that is with CanSpend(), we want to check that even if the time arrives when user mode changes to credit, we are still able to deduct funds since the cash availability differs between credit and debit mode

Fix Review: Acknowledged

Improved max spend functions on Cash

Project Overview

This report presents the findings of a manual code review for the **Improved max spend function changes** audit within the **EtherFi Cash** project. The work was undertaken from **June 11th to June 13th 2025**

The following contract list is included in the scope of this audit:

- src/interfaces/ICashModule.sol
- src/modules/cash/CashLens.sol

The code modifications examined during this review were implemented in the following pull request - PR#30

Informational Issues

I-01. Discrepancy between dev comments and actual implementation

Description: According to the comments describing the behavior of getMaxSpendDebit():

".. if a token has zero value, subsequent tokens won't be available for spending even if the deficit is covered"

However in the current code tokens that have 0 value are just skipped, so that the next token is used to cover the deficit. First inside _calculateTokenValues() if the effectiveBalance is 0 the loop just continues to the next token, then the same thing happens in _coverDeficitAndCalculateSpendable(). So if we have tokenValuesInUsd array like this - [0,0,50] and deficit of 30, the last token value would still be spent even though the first have value of 0 and in the end totalSpendableUsd would evaluate to 20

Impact: Potential inconsistency with the expected behavior of getMaxSpendDebit()

Recommendations: Update the comments to reflect the actual behavior of the function

Customer's response: Fixed in commit 7dc20a4

Fix Review: Fixed

I-02. Incorrect example in function description

Description: The @custom:example Underwater Position description tag for getMaxSpendDebit() describes an example scenario of how the function should work, however the example is not correct.

```
JavaScript
* @custom:example Underwater Position
     * // Safe: $1000 USDC (80% LTV), $500 USDT (80% LTV), $800 borrowings, $600 max borrow
     * // Deficit: $200, needs $250 collateral at 80% LTV from USDC (first preference)
     * // Returns: spendableAmounts=[750e6, 500e6], amountsInUsd=[750e18, 500e18],
total=$1250
     *
```

The correct version would look like this:

```
JavaScript
* @custom:example Underwater Position
     * // Safe: $1000 USDC (80% LTV), $500 USDT (80% LTV), $1400 borrowings, $1200 max borrow
     * // Deficit: $200, needs $250 collateral at 80% LTV from USDC (first preference)
     * // Returns: spendableAmounts=[750e6, 500e6], amountsInUsd=[750e18, 500e18],
total=$1250
```

The total value of the USDC+USDT is 1500, meaning the max borrow is 1200 (80%), so borrowings should not be 800 (healthy) but above 1200

Impact: Inconsistent example scenario

Recommendations: Update the example with the correct values

Customer's response: Fixed in commit 7dc20a4

Fix Review: Fixed

I-03. Ineffective rounding

Description: getMaxSpendDebit() executes the following arithmetics in attempt to round down to 4 decimals (here & here, as described in the comments), but the calculation is redundant and does not affect the value in any way (it remains the same):

```
JavaScript
(totalValueInUsd * 10 ** 4) / 10 ** 4
```

totalValueInUsd reflects the accumulated prices of each token returned by the price oracle. Those prices are in 6 decimals precision, so in order to reduce them to 4 decimals this is the correct calculation:

```
JavaScript
(totalValueInUsd * 10 ** 4) / 10 ** 6 // (remove oracle precision)
```

Be mindful that this will lose the value from last two digits that get trimmed

Impact: Improper decimal conversion

Recommendations: Consider using the the 6 decimals precision from the oracle instead of reducing it further or if that is necessary use the above calculation

Customer's response: Fixed in commit 7dc20a4

Fix Review: Fixed

Scroll ERC20 Bridge Adapter

Project Overview

This report presents the findings of a manual code review for the **Scroll ERC20 Bridge Adapter** audit within the **EtherFi Cash** project. The work was undertaken from **June 16th to June 18th 2025**

The following contract list is included in the scope of this audit:

- src/top-up/bridge/ScrollERC20BridgeAdapter.sol

The code modifications examined during this review were implemented in the following pull request - PR#32

Informational Issues

I-01. Excess ETH left from the bridging fee would remain in the TopUpFactory

Description: Bridging funds through the L1GatewayRouter requires paying a fee in ETH. Fees fluctuate based on activity and ScrollERC20BridgeAdapter uses a hardcoded gasLimit which would include an extra buffer, to make sure that there would always be enough funds to pay for bridging, even in periods of high activity.

After bridging Scroll refunds all leftover ETH(provided for the fee), back to the sender, which is TopUpFactory, however the ETH is provided by the caller(msg.sender) of bridge(). This means that the returned assets would go to TopUpFactory, not the actual payer of the fee.

Recommendations: The leftover amount would probably be small (the difference between the hardcoded fee amount sent and what was actually paid) and will accumulate in TopUpFactory. There is no serious outcome from this, other than the TopUpFactory receiving a small ETH donation.

Given the small amounts of ETH that would be in excess and that most of the time the bridging logic would be executed (fees will be paid) by EtherFi controlled nodes it should be ok to leave it like this and just add a comment to the function explaining that the excess ETH will not get reimbursed.

However a good optimization to TopUpFactory.bridge() would be to check that msg.value is not > than bridgeFee - which would make sure that any ETH beyond the required for bridging would not get locked (in case the caller sends more ETH than necessary)

Customer's response: Acknowledged

Fix Review: Acknowledged

Cashback upgrade

Project Overview

This report presents the findings of a manual code review for the **Cashback upgrade** audit within the **EtherFi Cash** project. The work was undertaken from **June 16th to June 18th 2025**

The following contract list is included in the scope of this audit:

- src/cashback-dispatcher/CashbackDispatcher.sol
- src/interfaces/*
- src/libraries/EnumerableAddressWhitelistLib.sol
- src/modules/cash/CashEventEmitter.sol
- src/modules/cash/CashModuleCore.sol
- src/modules/cash/CashModuleSetters.sol
- src/modules/cash/CashModuleStorageContract.sol

The code modifications examined during this review were implemented in the following pull request - PR#31

Low Severity Issues

L-01 Inconsistent update of totalCashbackEarnedInUsd state variable

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: CashModuleCore.sol	Status: Fixed	

Description: Inside _cashback() of CashModuleCore the totalCashbackEarnedInUsd state variable is updated only when the call to cashbackDispatcher succeeds(the try clause) but not in case of failure(catch clause). A fail (like in the case of the dispatcher being paused) does not change the fact that a cashback has been earned for that account (claimable at a later stage).

The variable is used only for informational purposes when fetching safe data through getData(), hence the low severity

Recommendations: Update totalCashbackEarnedInUsd even if the call to the dispatcher fails

Customer's response: Fixed in commit 02956e03

Fix Review: Fixed

Informational Issues

I-01. Initializer would revert if called a second time

Description: Part of the changes in CashbackDispatcher include updates in the initialize() function, where now an array of tokens is provided to configure the supported cashback tokens. This is an update from the previous single token model and would require calling initialize() in order to set the new token configuration.

However if the contract was already deployed and the initializer invoked in its previous version (single cashback token version), then calling initilize() a second time to upgrade it would revert, since initializers can be called only once.

Recommendations: If the new logic has to be applied on top of an already deployed and initialized instance of CashbackDispatcher, make sure to use a reinitializer (available through the already inherited OZ UUPSUpgradeable) instead of the already spent initializer. If it is an entirely new deployment, than it is ok to use the current initializer

Customer's response: Acknowledged - *"Tokens will be configured through the new configureCashbackToken() function"*

Fix Review: Acknowledged

I-02. Extra validation

Description:

- Consider adding a sanity check inside cashback() of CashbackDispatcher that makes sure the provided recipient is not set to address(0). The function is only called by the cashModule which does not validate the provided recipient address as well
- Inside getPendingCashback() of CashModuleCore check that the tokens array does not contain duplicate values. The same recommendation applies to clearPendingCashback() for the user & token arrays

Recommendations: Consider implementing the above checks

Customer's response: Fixed in commit 02956e03 & 9c44833

Fix Review: Fixed

I-03. Unwhitelisting cashback tokens can be blocked if they were disabled in the price provider first

Description: The configureCashbackToken() is called by CASHBACK_DISPATCHER_ADMIN_ROLE inside the CashbackDispatcher to un/whitelist tokens to be used for cashbacks:

```
JavaScript
for (uint256 i = 0; i < len; ) {
            //@audit-info - maybe check this only for token whitelisting
            if ($.priceProvider.price(tokens[i]) == 0)
                revert CashbackTokenPriceNotConfigured();
            unchecked {
                ++i;
            }
    }
```

There is a requirement that the token is still configured in the price provider. However it makes sense to skip this check when unwhitelisting , since it would block a token from being disabled as cashback, even when it already was disabled in the priceProvider.

Recommendations: Consider checking the prices provider configuration only for tokens that are being whitelisted

Customer's response: Fixed in commit 02956e03

Fix Review: Fixed

I-04. Use an already defined variable for consistency

Description: Inside _cashback() of the CashModuleCore in the following line use the already defined amountInUsd variable in the beginning of the loop for consistency and efficiency

Recommendations: Consider the above recommendation

Customer's response: Fixed in commit 02956e03

Fix Review: Fixed

Cash Module Additional Audit Round

Project Overview

This report presents the findings of an additional manual code review round conducted on the **EtherFi Cash** project. The work was undertaken from **May 2nd to May 8th 2025**

The following contract list is included in the scope of this audit:

- `src/beacon-factory/*`
- `src/cashback-dispatcher/*`
- `src/data-provider/*`
- `src/debt-manager/*`
- `src/hook/*`
- `src/libraries/*`
- `src/modules/*`
- `src/oracle/*`
- `src/safe/*`
- `src/role-registry/*`
- `src/settlement-dispatcher/*`
- `src/top-up/*`
- `src/utils/*`

High Severity Issues

H-01 Improper debt conversion during liquidation

Severity: **High**	Impact: **High**	Likelihood: **High**
Files: DebtManagerCore.sol	Status: Fixed	

Description: The liquidation flow in DebtManagerCore.sol is executed through the liquidate() function. It is done in 2 steps - it first tries to liquidate half of the user collateral and if the account is still liquidatable it liquidates the rest of the available collateral in the safe.

```JavaScript
function _liquidateUser(
        ...
    ) internal {
        DebtManagerStorage storage $ = _getDebtManagerStorage();

        uint256 debtAmountToLiquidateInUsd = _getActualBorrowAmount(
            $.userNormalizedBorrowings[user][borrowToken].ceilDiv(2),
            interestIndex
        );
        _liquidate(
            ...
            debtAmountToLiquidateInUsd, <---
            interestIndex
        );

        if (liquidatable(user))
            _liquidate(
               ...
                $.userNormalizedBorrowings[user][borrowToken], <----
                interestIndex
            );
```

```
    }
```

The USD value of the assets borrowed by a safe are saved to a mapping called userNormalizedBorrowings, where the amounts are stored in their normalized version (e.g. without the accrued interest). When debt is paid or liquidated its value is increased by the accrued interest to reflect the actual amount owed to the protocol. To get the normalized amount with accrued interest the _getActualBorrowAmount() is used and to convert it back to the raw amount _getNormalizedAmount().

The issue here is that the conversion of normalized amounts is not done properly, leading to improper debt accounting:

- Before the first liquidation attempt half of the normalized debt is converted through _getActualBorrowAmount() and provided to the internal _liquidate(). The function itself calculates the actual amount that was liquidated and finally calls _getNormalizedAmount() to convert it back to the raw debt amount and update the mapping
- The problem is that the second time _liquidate() is called the normalized debt is not converted to actual amounts but provided directly as normal amounts. Regardless _liquidate() would still call _getNormalizedAmount() on it, assuming it was converted before that, which is not the case

Impact: The amounts in the second liquidation attempt would be improperly calculated, since they would use normalized rather than actual values, while the code acts upon the assumption that actual values have been provided - for example the accrued interest will not be paid, only the original amount

Recommendations: Call _getActualBorrowAmount() on the normalized user borrowings before providing them to the _liquidate() function.

Customer's response: Fixed in commit 64bf6de

Fix Review: Fixed

Medium Severity Issues

M-01 Improper liquidation bonus calculation

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: DebtManagerCore.sol	Status: Fixed	

Description: When we liquidate a user, there is a condition where the bonus is calculated on the total collateral amount for a token instead of the amount being liquidated which gives the liquidator some more bonus than expected

```
JavaScript

....
uint256 totalCollateral = IERC20(collateralToken).balanceOf(user);
uint256 maxBonus = (totalCollateral *
$.collateralTokenConfig[collateralToken].liquidationBonus) / HUNDRED_PERCENT;
...
```

Recommendations: Calculate the bonus based on the actual amount liquidated, not the entire collateral

Customer's response: Fixed in commit e1f27a2

Fix Review: Fixed

M-02 Debt supplier funds can get locked in the manager

Severity: **Medium**	Impact: **Medium**	Likelihood: **Low**
Files: DebtManagerCore.sol	Status: Fixed	

Description: DebtManager is the contract used by Safes to borrow tokens. Liquidity in the manager is provided through the supply() function. Each supplier receives shares that reflect the amount deposited relative to the total amount of deposits.

There is also a protection against inflation attacks through the minShares variable, which enforces a minimum amount of shares that must be created initially for that token in the manager in order for the deposit to be accepted.

Withdrawing assets happens through the withdrawBorrowToken() function, which makes sure that after the withdrawal the amount of shares left is not below the minimum

```JavaScript
function withdrawBorrowToken(
        address borrowToken,
        uint256 amount
    ) external whenNotPaused {
    ...
    if (
            sharesLeft != 0 &&
            sharesLeft < $.borrowTokenConfig[borrowToken].minShares
        ) revert SharesCannotBeLessThanMinShares();
     ...
}
```

This requirement creates a scenario where the last depositors for a token in the vault might not be able to withdraw all their assets.

Exploit Scenario: Here is an example of the issue:

DebtManager has 2 depositors Bob and Alice - each has deposited 500 tokens and received 500 shares in return. The total balances of the Manager are 1000 tokens and minShares is configured to 500 shares. This is what can happen:
- Alice withdraws 250 tokens successfully (burns 250 shares)
- Bob withdraws 250 tokens successfully (burns 250 shares)
- Alice tries to withdraw tokens but fails - since available shares are 500 which equals minShares, any further withdraws fail. Same thing applies for Bob
- None of them hold all shares in order to reduce them to 0, which causes them to lock each other funds

Recommendations: Currently this is an inherent issue to the min shares design. One approach could be to implement the shares restriction per account, instead of per token. This way there would still be some protection from inflation attacks, but also each depositor would be able to burn all his shares

Customer's response: Fixed in commit c08b741

Fix Review: Fixed

M-03 No check for duplicate tokens in canSpend causes improper debt calculation

Severity: **Medium**	Impact: **High**	Likelihood: **High**
Files: CashLens.sol	Status: Fixed	

Description: The canSpend() function does not check if the provided tokens array contains duplicates. If such an array is provided then the _getCollateralBalanceWithTokensSubtracted() logic executed further down the call stack will only consider the first token occurrence. So if the array contains tokenA 2 times, where the first occurrence has a spend value of 1 and the second a spend value of 1000, then canSpend() would evaluate the borrowing power as if only 1 token was spent (instead of 1001)

```
JavaScript
function _getCollateralBalanceWithTokensSubtracted(
        ....
    ) internal view returns (IDebtManager.TokenData[] memory, string memory) {
...

       if (safeData.mode == Mode.Debit) {
             // Only considers first occurance
               for (uint256 j = 0; j < tokens.length; j++) {
                    if (collateralTokens[i] == tokens[j]) {
                    ....
                    }
                }
             }
....
}
```

Recommendations: Make sure to check that the tokens array does not contain duplicate values

Customer's response: Fixed in commit 0f3cd19

Fix Review: Fixed

Low Severity Issues

L-01 Token deduplication not implemented upon spending

Severity: **Low**	Impact: **Medium**	Likelihood: **Medium**
Files: CashModuleCore.sol	Status: Fixed	

Description: Provided token array when calling spend() is not validated to contain duplicate values for debit mode spending

Recommendations: When spending make sure to check that the tokens input does not contain duplicate elements

Customer's response: Fixed in commit 0f3cd19

Fix Review: Fixed

L-02 Callback from cashback token to retrieve pending cashback can drain cashback tokens

Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: CashModuleCore.sol	Status: Fixed	

Description: In case the cashbackToken configured in CashBackDispatcher allows reentrant behaviour (causing external calls during transfer()) it will allow a receiver to drain the CashModule through _retrievePendingCashback(), which updates the state only after the token transfer

```javascript
function _retrievePendingCashback(address user) internal {
        CashModuleStorage storage $ = _getCashModuleStorage();

        if ($.pendingCashbackInUsd[user] != 0) {
             // @audit -re-enter
            (address cashbackToken, uint256 cashbackAmount, bool paid) =
$.cashbackDispatcher.clearPendingCashback(user);
            if (paid) {
                $.cashEventEmitter.emitPendingCashbackClearedEvent(user, cashbackToken,
cashbackAmount, $.pendingCashbackInUsd[user]);
                delete $.pendingCashbackInUsd[user]; //@audit state is updated AFTER transfer
            }
        }
    }
```

Recommendations: Add reentrancy guard to clearPendingCashback() & spend().

Customer's response: Fixed in commit fdfdbb4 & 68b9f1a

Fix Review: Fixed

L-03 Normalized amount calculation should always round up

Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: DebtManagerCore.sol	Status: Fixed	

Description: The _getNormalizedAmount() call in borrow() should always round up in favor of the protocol, not the user

```
JavaScript
function borrow(BinSponsor binSponsor, address token, uint256 amount) external whenNotPaused
onlyEtherFiSafe {
...
uint256 normalizedAmount = _getNormalizedAmount(borrowAmt, newInterestIndex);
...
}

...

function _getNormalizedAmount(uint256 actualAmount, uint256 interestIndex) internal pure
returns (uint256) {
        return actualAmount.mulDiv(PRECISION, interestIndex, Math.Rounding.Floor);
    }
```

Recommendations: Round up when calling _getNormalizedAmount() in borrow()

Customer's response: Fixed in commit 38a15b5

Fix Review: Fixed

L-04 Max can spend returns 0 if withdrawal exists for a token

Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: CashLens.sol	Status: Fixed	

Description: maxCanSpend() improperly calculates the maximum amount available to spend in debit mode by using the total token balanceOf() of a Safe, instead of using only the available assets.

As result _getCollateralBalanceWithTokensSubtracted() would always return 0, since it uses the available balances to compare it with the provided amount (which is always the full balanceOf)

Recommendations: Before calling _getCollateralBalanceWithTokensSubtracted() inside _calculateDebitModeAmount() make sure to subtract the pending withdrawal amounts from the total balances

Customer's response: Fixed in commit 344c5f9

Fix Review: Fixed

I-01. Unreachable code

Description: Unnecessary withdrawal cancellation in _spendDebit(). The only scenario where there might be a revert is if there are not enough assets in the Safe. So in that case it doesn't make sense to cancel the withdrawal in the catch clause

Recommendation: Do not wrap the transfer in a try clause

Customer's response: Fixed in commit a49d891

Fix Review: Fixed

I-02. Unnecessary division

Description: The arithmetics at the end of the _calculateCreditModeAmount() in CashLens do not change anything and should be removed

Recommendation: Remove unnecessary multiplication and division

Customer's response: Fixed in commit 149f14d

Fix Review: Fixed

I-03. Use total spend while giving cashback in credit mode

Description: In this line of code consider using totalSpendingInUsd instead of amountsInUsd[0]. It does not affect the code any way, but it is more concise and consistent with the rest of the logic in that function

Recommendation: Use totalSpendingInUsd instead of amountsInUsd[0]

Customer's response: Fixed in commit f113dca

Fix Review: Fixed

I-04. Referrer cashback is not capped to MAX_CASHBACK_PERCENTAGE

Description: There is no restriction on the max value of the parameter provided to setReferrerCashbackPercentageInBps()

Recommendation: Restrict the max amount to MAX_CASHBACK_PERCENTAGE, instead of HUNDRED_PERCENT_IN_BPS

Customer's response: Fixed in commit 6c8f563

Fix Review: Fixed

Newly uncovered findings

Project Overview

This report presents the findings of an additional manual code review round conducted on the **EtherFi Cash** project. The work was undertaken from **May 4th to May 18th 2025**

Note: The findings outlined in this section are new and require a fixes response from the ether.Fi team

The following contract list is included in the scope of this audit:

- `src/beacon-factory/*`
- `src/cashback-dispatcher/*`
- `src/data-provider/*`
- `src/debt-manager/*`
- `src/hook/*`
- `src/libraries/*`
- `src/modules/*`
- `src/oracle/*`
- `src/safe/*`
- `src/role-registry/*`
- `src/settlement-dispatcher/*`
- `src/top-up/*`
- `src/utils/*`

Medium Severity Issues

M-01 Malicious actor can grief interest index accrual

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: DebtManagerStorageContract.sol	Status: Acknowledged	

Description: The debt manager uses an interest index that auto increments every second and it is the mechanism to accrue interest upon the borrowed funds. Below is the logic for calculating index accrual

```javascript
function _updateInterestIndex(
        address borrowToken
    ) internal returns (uint256) {
    ....
        uint256 currentIndex = config.interestIndexSnapshot;
        config.interestIndexSnapshot = getCurrentIndex(borrowToken);
        config.lastUpdateTimestamp = uint64(block.timestamp);
    ....
}
...
function getCurrentIndex(
        address borrowToken
    ) public view returns (uint256) {
       ...
        uint256 timeElapsed = block.timestamp - config.lastUpdateTimestamp;
        ...
        uint256 interestAccumulated = config.interestIndexSnapshot.mulDiv(
            config.borrowApy * timeElapsed,
            HUNDRED_PERCENT
        );
        return config.interestIndexSnapshot + interestAccumulated;
    }
```

Depending on the configured parameters it is possible that a malicious actor exploits the formula and the logic of interest accrual and cause interestIndexSnapshot to stop incrementing and as result stop interest accrual for extended periods at low cost

Exploit Scenario: Here is an example with real configuration values from the protocol:

- Based on the SupportBorrowToken script defined in the scripts folder the liquidUsd & eUsd tokens are configured with borrowAPY of 1
- interestIndexSnapshot is always initialized with a value of PRECISION(1e18)
- Block time in Ethereum is ~ 12 seconds, so timeElapsed between 2 blocks is around that time

  Here is how the index accrual formula looks based on the above parameters:

  interestIndexSnapshot * (apy* timeElapsed)/HUNDRED_PERCENT => 1e18 * (1*12) / 100e18 => 0

As a result , if someone invokes _updateInterestIndex() on each block (calling repay() with dust amounts) in the beginning, he can delay it for as long as he likes since the snapshot would always be incremented by 0. It is important to note that the calculated index is still 0 for longer periods - like 1 minute (60). Anything below 100 seconds(~ 8 blocks) with the current configuration rounds down to 0.

Recommendations: Inside _updateInterestIndex() make sure to check if getCurrentIndex() returns the same snapshot as the already saved one. If that is the case it means there was no interest accrued and lastUpdateTimestamp should not be updated yet - this would allow enough time to accumulate and prevent updates before sufficient interest has accumulated.

Customer's response: Acknowledged - *"We think it's fine for now because the interest rate is 0. We don't plan to increase it anytime soon. We are going to transition to an Aave as debt manager model soon probably."*

Fix Review: Acknowledged

M-02 Sending max amounts will not work in the AAVE module for ETH

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AaveV3Module.sol	Status: Fixed	

Description: Inside the _repay() function there is a check if the provided amount is type(uint256).max and if that is the case it updates the amount to the full actual amount owed by the safe by calling getTokenTotalBorrowAmount()

```
JavaScript
function _repay(...){
       ...
       if (amount == type(uint256).max) {
            if (asset == ETH) getTokenTotalBorrowAmount(safe, weth); //@audit amount not
updated
            else amount = getTokenTotalBorrowAmount(safe, asset);
        }
}
```

The problem is that if the asset is ETH, the amount is not updated with the result of getTokenTotalBorrowAmount(). As consequence the provided amount to IAavePoolV3.repay() remains type(uint256).max which would always cause a revert due to the next line:

```
JavaScript
if (asset == ETH) bal = safe.balance;
 ....
(bal < amount) revert InsufficientBalanceOnSafe();
```

Impact: Repayments with type(uint256).max as amount would always fail for ETH

Recommendations: Make sure to re-assign the amount variable with the result of getTokenTotalBorrowAmount() as in the else clause.

Customer's response: Fixed in commit 678ee11

Fix Review: Fixed

Low Severity Issues

L-01 Malicious actor can grief the TopUp Factory by causing more fees to be paid than necessary

Severity: **Low**	Impact: **Medium**	Likelihood: **Medium**
Files: TopUpFactory.sol	Status: Fixed	

Description: TopUpFactory is used for two important tasks. To transfer tokens from the TopUp contracts to itself and to bridge those assets through configured bridge adapters. Each bridging transaction in ETH requires a fee to be withheld from the transferred amount, that is paid to the adapter.

Currently anyone can transfer balances from any registered TopUp contracts into TopUpFactory by calling processTopUp() or processTopUpFromContracts() which are permissionless. Bridging is also permissionless.
This creates an opportunity for an account to execute those flows in a suboptimal manner and cause the Factory to pay more fees than it normally would

```JavaScript
function bridge(address token) external payable whenNotPaused {

        ...
        uint256 balance = token == ETH
             ? address(this).balance - 0.01 ether
             : IERC20(token).balanceOf(address(this));
        ...
}
```

Exploit Scenario: Here is a concrete scenario:
- Factory has 10 deployed TopUp contracts each having 2 ETH in them and the Factory ETH

balances are currently 0
- Bob is malicious and does the following - calls processTopUp() only for 1 TopUp contract and then calls bridge() paying the bridging fee. Bob repeats this 10 times for each TopUp.
- As a result the transaction fee is paid 10 times - x10 more fees. In comparison the normal flow would be to call processTopUp() for all 10 TopUps at once and then bridge the cumulative amount at once, paying the bridging fee only once

Recommendations: Given the permissionless nature of the functions it would be hard to prevent this issue. Consider if those functions could be protected with a role, or at least the bridge() function.

Another approach might be to take the bridging fee from the caller of bridge(), instead of deducting it from the TopUp amounts that were transferred. This will make the exploit infeasible, since the exploiter would be the one to pay for the fees

Customer's response: Fixed in commit ded8cde

Fix Review: Fixed

L-02 ETH is not recoverable		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: TopUpFactory.sol	Status: Fixed	

Description: TopUpFactory() implements a recoverFunds() function that allows admins to transfer out ERC20 tokens from the contract. However the function does not support recovering ETH balances and since the contract also uses ETH it makes sense to also make it recoverable

Recommendations: Consider making ETH recoverable as well

Customer's response: Fixed in commit 9dfb26a

Fix Review: Fixed

L-03 Anyone can prevent borrow tokens from getting disabled

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: DebtManagerAdmin.sol	Status: Acknowledged	

Description: The unsupportBorrowToken() function is called by DEBT_MANAGER_ADMIN_ROLE to remove currently active borrow tokens. The function requires that no balances and borrows exist for the token. Since the check depends on IERC20(borrowToken).balanceOf(address(this)) it makes it very easy for anyone to deposit 1 wei and DOS the function

Impact: Borrow tokens are prevented from being disabled

Recommendations: One approach is to switch to internal accounting to track the deposited assets, however this would require multiple changes throughout the contract, which is risky. Consider if it makes sense to remove the balance check and use the function only in emergency scenarios where borrowed tokens need to be disabled immediately.

Customer's response: Acknowledged

Fix Review: Acknowledged

L-04 Oracle prices for stable tokens are prone to value extraction

Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: PriceProvider.sol	Status: Acknowledged	

Description: The _getStablePrice() function is used to handle calculation of tokens that have been configured as stable by the protocol admins. The specific detail in handling those type of tokens is how their price is derived:

```
JavaScript
function _getStablePrice(
      ...
    ) internal pure returns (uint256) {
        ....
        if (
            uint256(_price) > STABLE_PRICE - MAX_STABLE_DEVIATION &&
            uint256(_price) < STABLE_PRICE + MAX_STABLE_DEVIATION
        ) return STABLE_PRICE;
        else return _price;
        ....
}
```

In case the returned price deviates by 1% below or above the hardcoded STABLE_PRICE (1e6) the returned price is STABLE_PRICE instead of the actual price. This could be a dangerous approach since it allows advantageous actors to extract value at the detriment of the protocol.

Here is an example:

- TokenA has been configured with isStableToken flag set to true
- 1 unit of TokenA actual price is 1010000(1.01$), which is exactly 1% above STABLE_PRICE(1e6), but the returned price would still be 1e6

- Bob is advantageous and does the following - borrows from DebtManager 10000 tokens, which the manager values at 10000$, when they actually cost 10100$.
- Bob swaps the 101000$ TokenA on external exchange for another **non**-stable collateral TokenB and deposits it to the safe. Out of thin air the collateral of Bob has been increased by 100$ - he can use it to borrow even more TokenA.

This is just one example, the exploit opportunities are numerous given the many different flows existing in the protocol.

According to the current scripts, there are no tokens configured with an isStableToken flag set to true, hence this issue is marked as Low. However the team must be aware of the risks associated with adding such tokens

Impact: MEV extraction due to price discrepancies on stable tokens

Recommendations: In case there is no significant consideration behind treating stable coins in a specific way, consider removing the deviation logic and always use the actual prices returned by the oracle

Customer's response: Acknowledged - "*We'll use this until transition to Aave model*"

Fix Review: Acknowledged

Informational Issues

I-01. Internal function not used

Description: The _getCollateralBalanceWithTokenSubtracted() function inside CashModuleCore.sol is not used.

Recommendation: Consider removing it

Customer's response: Fixed in commit 5f1e62a

Fix Review: Fixed

I-02. getUserTotalCollateral might include zero balance tokens

Description: When calling getUserTotalCollateral() the loop validates if balances for a token are zero and skips including them in the returned array - that's in the first if (balance != 0) check. Right after that the balance is decreased by the pendingWithdrawalAmount - the code does not check if the left over balance is above zero and includes it always in the returned array, which is not consistent with the logic of the function

Recommendation: After deducting pendingWithdrawalAmount check once again if the balances are not 0

Customer's response: Fixed

Fix Review: Fixed in commit 390fe92

I-03. Inconsistent ride bus logic when bridging through Settlement dispatcher

Description: The contract is used to bridge tokens through Stargate to other chains. Each token bridging parameters are configured by trusted roles. However in the logic of prepareRideBus() there is an inconsistency in how tokens are handled, which might be problematic.

The issue is that prepareRideBus() always expects ERC20 tokens and checks their balances as such. At the same time at the bottom the code assumes that stargate bridges using ETH can be used as well (token() == address(0x0)). But given that only ERC20 tokens are handled it is clear that ETH stargate bridges cannot be used currently.

```
JavaScript
function prepareRideBus(
        address token,
        uint256 amount
    )

       ...
       if (token == address(0) || amount == 0) revert InvalidValue();
        //@audit - if Stargate token is ETH, then this won't work
        if (IERC20(token).balanceOf(address(this)) < amount)
            revert InsufficientBalance();
       ...
       if (IStargate(stargate).token() == address(0x0)) {
            valueToSend += sendParam.amountLD;
        }

}
```

For reference the same issue has already been handled in StargateAdapter.sol for top ups, where WETH is converted before bridging through a Stargate ETH bridge

Impact: Configuring for stargate ETH bridges would not work

Recommendations: Overall using plain ETH is risky and not advisable. It would be better to use only ERC20 tokens (WETH). If ETH is still intended to be used in the dispatcher, then consider implementing the approach from the adapter

Customer's response: Fixed in commit 28b7e84 & 7b88f5d

Fix Review: Fixed

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.