CODESPECT

BetterBank contracts

SECURITY ASSESSMENT REPORT

October 1, 2025

Prepared for:

BetterBank

Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the security assessment conducted by CODESPECT for the smart contracts of BetterBank. BetterBank is a revolutionary DeFi protocol that reimagines lending, borrowing, and wealth generation.

The scope of this audit covers the custom BetterBank contracts. As the BetterBank forked AAVE-v3 codebase, the review focus only on the custom contracts which interacts with the AAVE part and brings new incentives and funcionality to the overall system.

The audit was performed using:

a) Manual analysis of the codebase.

b) Dynamic analysis of smart contracts, execution testing.

CODESPECT found twelve points of attention, six classified as `Medium`, two classified as `Low`, three classified as `Informational`, and one classified as `Best Practices`. All of the issues are summarised in Table 2.

Organisation of the document is as follows:

- **Section 5** summarizes the audit.
- **Section 6** describes the functionality of the code in scope.
- **Section 7** presents the issues.
- **Section 8** presents additional issues found during the fix review phase.
- **Section 9** presents the additional notes of the auditors.
- **Section 10** presents centralisation risk and trust assumptions.
- **Section 11** discusses the documentation provided by the client for this audit.
- **Section 12** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
Medium	0	6	0
Low	0	1	1
Informational	0	0	3
Best Practices	0	1	0
Total	**0**	**8**	**4**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	BetterBank
Type of Project	Lending and Borrowing protocol
Duration of Engagement	7 Days
Duration of Fix Review Phase	2 Days
Draft Report	September 24, 2025
Final Report	October 1, 2025
Repository	BB-Custom-contracts
Commit (Audit)	43fba5ed6a5f47d6ae997660a7fe656c57f44b30
Commit (Final)	f484e550e31e8d9775bfb49db423c04bb49bbb5a
Documentation Assessment	Medium
Test Suite Assessment	Medium
Auditors	jecikPo, namx05, 0xAdityaRaj

Table 3: Summary of the Audit

5.1 Scope - Audited Files

	Contract	LoC
1	Zapper.sol	317
2	FavorTreasury.sol	261
3	PulseMinter.sol	221
4	LPOracle.sol	190
5	Staking.sol	182
6	Favor.sol	138
7	MinterOracle.sol	101
8	UniTWAPOracle.sol	97
9	Epoch.sol	75
10	Esteem.sol	24
11	interfaces/IFavorToken.sol	7
12	interfaces/PriceProvider.sol	6
13	interfaces/BBToken.sol	6
14	interfaces/IOracle.sol	6
15	interfaces/ITreasury.sol	6
16	interfaces/IMasterOracle.sol	5
17	interfaces/IBasisAsset.sol	4
18	interfaces/IGrove.sol	4
	Total	**1650**

5.2 Findings Overview

	Finding	Severity	Update
1	Flash loan inflatable staking balance allows stealing rewards	Medium	Fixed
2	Missing slippage protection when interacting with UniswapV2 Pair	Medium	Fixed
3	Pausing of `FavorTreasury` queues outstanding epochs to be executed	Medium	Fixed
4	Staking user can inflate RPS on low total supply of shares	Medium	Fixed
5	UniswapV2 price TWAP accumulator underflow will lead to oracle update failure	Medium	Fixed
6	Zapper can retain dust affecting depositor funds	Medium	Fixed
7	Lack of `whenNotPaused` modifier on `claimReward`	Low	Fixed
8	Normalization function can underflow with tokens having more than 18 decimals	Low	Acknowledged
9	CREATE2 vulnerability in favor token allows sell tax bypass	Info	Acknowledged
10	Protocol not prepared to handle base tokens with fee-on-transfer	Info	Acknowledged
11	`UniTWAPOracle` and `LPOracle` state can become stale if it is not updated	Info	Acknowledged
12	Usage of two-step ownership transfer is recommended	Best Practices	Fixed

5.3 Findings Overview - Fix Review Phase

	Finding	Severity	Update
1	Flash loan functionality in `LPZapper` always reverts	Medium	Fixed

6 System Overview

BetterBank is a multi-token ecosystem with seigniorage mechanics that combines DeFi yield farming, lending, and staking functionality across PulseChain. The system operates with two primary tokens (Esteem and Favor) and multiple specialized contracts handling minting, staking, treasury operations, and automated market-making through integrated DEX and lending protocols.

- **Esteem** – The primary utility token serving as the ecosystem's backbone. Used for staking in Groves (staking pools) to earn Favor rewards and as the redemption mechanism for Favor tokens.
- **Favor** – Base trading assets paired with underlying tokens (PLS, PLSX, DAI). Features a dynamic taxation system with up to 50% sell tax and automated buy bonuses. Transfers to non-tax-exempt contracts are automatically treated as sells and taxed accordingly.
- **PulseMinter** – Handles Esteem token minting and Favor redemption operations. Integrates with multiple price oracles and automatically deposits portion of proceeds into BetterBank lending pools while allocating portion to the team's treasury.
- **Staking (Grove)** – Implements snapshot-based reward distribution for Esteem stakers. Users stake Esteem tokens to earn newly minted Favor rewards distributed proportionally. Maintains up to 50,000 historical snapshots for gas optimization and allows penalty-free staking/unstaking.
- **FavorTreasury** – Automated seigniorage system operating on short time epochs. Calculates dynamic supply expansion based on TWAP price relative to paired tokens. Mints new Favor supply and distributes to respective Grove contracts for staker rewards.
- **Zapper** – Provides tax-exempt trading and liquidity management through a UniswapV2–style router integration and BetterBank lending pools. Implements flash loan functionality for leveraged liquidity provisioning and handles buy/sell operations with automatic bonus calculations and tax collection.

6.1 Token Flow Process

6.1.1 Esteem Minting

Users mint Esteem tokens via the `PulseMinter` contract using PLS, PLSX, or DAI at the current rate:

```
function mintEsteemWithPLS(uint256 deadline) external payable;

function mintEsteemWithToken(uint256 amount, address token, uint256 deadline) external;
```

Proceeds are automatically split with larger portion deposited into BetterBank lending pools and smaller sent to team treasury. An additional treasury bonus is minted to the protocol multisig on top of the user's minted amount.

6.1.2 Favor Token Mechanics

Favor tokens implement dynamic taxation where transfers to contracts trigger automatic sell taxation. Buy bonuses are calculated when TWAP falls below certain threshold:

$$\text{EsteemBonus} = \frac{\text{USDValue} \times \text{BonusRate}}{\text{EsteemPrice}}$$

$$\text{TreasuryBonus} = \text{EsteemBonus} \times 25\%$$

The `Zapper` contract serves as the primary tax-exempt interface for legitimate trading operations, allowing users to provide liquidity and earn yield without automatic taxation.

6.1.3 Staking & Rewards

Users stake Esteem in Grove contracts to earn Favor rewards distributed through Favor treasury:

```
function stake(uint256 amount) external;

function claimReward() external;
```

Rewards are calculated using snapshot-based distribution ensuring fair allocation based on stake duration and size. The reward formula is:

$$\text{UserReward} = \frac{\text{UserStake} \times (\text{LatestRPS} - \text{UserLastRPS})}{10^{18}} + \text{PreviousEarned}$$

Where RPS represents Reward Per Share accumulated over time.

6.2 Seigniorage Distribution

The `FavorTreasury` operates on hourly epochs, calculating expansion rate based on current TWAP price. New Favor tokens are minted and sent to respective Grove contracts for distribution to stakers. The circulating supply calculation excludes:

- Balances of excluded protocol addresses
- Favor tokens held in LP positions by excluded addresses
- Treasury and team multisig holdings

6.3 Oracle Integration

The system integrates multiple oracle types for accurate price feeds:

- `MinterOracle` – Primary price provider integrating Fetch Protocol for external feeds and coordinating TWAP oracles for tokens without Fetch feeds
- `UniTWAPOracle` – Time-weighted average pricing for Favor tokens to prevent manipulation and ensure price stability
- `LPOracle` – Geometric mean pricing for LP token valuations using reserve averaging to accurately calculate LP token prices in USD
- `Epoch` – Standardized time period enforcement for TWAP calculations across oracle contracts

All oracles implement price cap mechanisms and data freshness requirements to ensure system stability.

6.4 Flash Loan Integration

The `Zapper` contract provides flash loan functionality through BetterBank lending pools, enabling users to create leveraged liquidity positions:

```
function requestFlashLoan(uint256 _amount, address _favorToken) external;
```

The process works as follows:

a. User supplies Favor tokens to the Zapper
b. Zapper calculates optimal base token amount needed for LP creation
c. Flash loan is requested for the base token amount
d. Favor and base tokens are added to liquidity pool
e. LP tokens are deposited as collateral in lending pool
f. Base tokens are borrowed to repay the flash loan
g. User receives leveraged liquidity position

6.5 Redemption Mechanism

Users can redeem Esteem tokens for Favor tokens at a 70% rate through the `PulseMinter`:

```
function redeemFavor(uint256 _esteemAmount, BBToken _favorToken) external;
```

The redemption calculation:

$$\text{FavorReceived} = \frac{\text{EsteemAmount} \times \text{EsteemRate}}{\text{FavorPrice}} \times 70\%$$

This architecture enables seamless DeFi operations while maintaining protocol security through integrated oracle pricing, automated treasury management, and controlled token emission mechanisms. The system's modular design allows for independent operation of each component while maintaining coherent economic incentives across the entire ecosystem.

7 Issues

7.1 [Medium] Flash loan inflatable staking balance allows stealing rewards

File(s): Staking.sol

Description: The `Staking` contract is used to distribute Favor token rewards to Esteem token stakers. The rewards are delivered to the `Staking` contract through the `allocateSeigniorage(...)` function which is called from `FavorTreasury`. Only approved addresses can call it, yet the `FavorTreasury.allocateSeigniorage(...)` (which calls `Staking.allocateSeigniorage(...)`) can be called by anyone, only once during epoch. The `Staking` contract doesn't have stake lockup period, users are allowed to stake and withdraw Esteem anytime.

Since anyone can call `FavorTreasury.allocateSeigniorage(...)` it makes the `Staking` contract vulnerable to a flash loan attack that could inflate a malicious user's stake by using the following sequence:

a. Malicious user takes large Esteem flash loan at the approaching end of the period;
b. The flash loan is staked at `Staking`;
c. The `FavorTreasury.allocateSeigniorage(...)` is called and smaller RPS is calculated because of the large stake;
d. User collects Favor rewards and repays the flash loan;

Impact: A malicious user could take large chunk of the Favor rewards at only the flash loan cost at the expense of the legitimate staker's rewards.

Recommendation(s): This could be solved in few ways:

a. Prevent staking and withdraw during same block or timestamp;
b. Enforce a small fee on the withdrawals;
c. Make the `FavorTreasury.allocateSeigniorage(...)` function admin/owner controlled only;

Status: Fixed

Update from BetterBank: Fixed in e0c58f4eae15d1e5579e15a0a64431db5d409499

7.2 [Medium] Missing slippage protection when interacting with UniswapV2 Pair

File(s): `Zapper.sol/L150` `Zapper.sol/L183`

Description: The `Zapper` contract allows handling of both base and Favor tokens - it manages adding liquidity on behalf of the user and can swap token to favor when needed. Both `_swap(...)` and `_addLiquidity(...)` rely on router functions but have hard-coded the minimum output parameters to `0`. In `_swap(...)`, the contract uses `router.swapExactTokensForTokensSupportingFeeOnTransferTokens(...)` with no slippage threshold, allowing execution at unfavorable exchange rates.

Impact: An attacker can front-run the swap with a sandwich attack. These omissions generally expose users to front-running and sandwich attacks, yet the presence of a transfer tax on Favor token reduces exploitability. Any attacker attempting to manipulate the reserves to force poor execution must pay the tax when moving Favor, which erodes profit margins and can make such attacks economically infeasible. Still, the lack of slippage protection means the system does not revert in adverse conditions, and users can unintentionally add liquidity or swap at unfavorable terms. The price could still be altered through adding liquidity which swaps half of the added tokens.

Recommendation(s): Enforce a configurable minimum token output when calling the router to ensure trades or liquidity adding reverts, if the expected rates cannot be met.

Status: Fixed.

Update from BetterBank: Fixed in b9413cc025d7f7cc76daf28224b917923daf732f

Update from CODESPECT: Lack of slippage protection in `_zapToken(...)` acknowledged.

7.3 [Medium] Pausing of `FavorTreasury` queues outstanding epochs to be executed

File(s): `FavorTreasury.sol`

Description: The `FavorTreasury` contract implements a custom epoch checkpoint mechanism instead of using the standard `Epoch.sol`. In this mechanism, functions with the `checkEpoch` modifier increment the epoch by **one** per call, rather than calculating the current epoch based on the elapsed time.

This design introduces unpredictable behaviour if the protocol is paused for an extended period. For example, if the protocol is paused for a week, calling `allocateSeigniorage(...)` after unpausing could process all outstanding epochs (e.g., 7 days × 24 = 168 epochs). This would mint `Favor` tokens for the entire paused duration. Additionally, an attacker could exploit this by taking a flash loan, staking via `Staking.sol`, and claiming rewards for all 168 epochs, potentially draining rewards intended for long-term stakers.

Impact:

- Unexpected minting of `Favor` tokens for paused periods;
- Potential manipulation of staking rewards;
- Risk of reward theft from long-term stakers;

Recommendation(s): Update the epoch logic to compute the current epoch based on the elapsed time rather than incrementing by one, similar to the approach in `Epoch.sol`.

Status: Fixed

Update from BetterBank: Was fixed, epoch system was reworked (1e7747aa89f79ff8ad55e8facc0a0e3a5cc3a699)

7.4 [Medium] Staking user can inflate RPS on low total supply of shares

File(s): `Staking.sol`

Description: On `Staking.sol` contract the `allocateSeigniorage(...)` function is used to add rewards. The RPS (Reward Per Share) value is calculated based on the amount deposited and total supply of shares:

```
uint256 nextRPS = prevRPS + ((amount * 1e18) / totalSupply());
```

If the total supply falls below `1e18` the RPS value is inflated to multiple times deposited amount.

While it is not likely that such a condition might occur, it may happen shortly before launch if a malicious user deposits a small amount just before initial rewards are loaded in. This will cause his RPS to grow significantly at the expense of later users.

Impact: While the occurence is not likely, as it would require specific timing (e.g. front-running initial rewards adding event), the impact would be severe as it the early deposit would get a disproportionate amount of rewards accrued. Those rewards in form of `favor` token, if claimed, would be at the expense of other stakers.

Recommendation(s): The `totalSupply() > 0` condition of the `allocateSeigniorage(...)` could be changed so that the total supply cannot be lower than `1e18`, yet it will prevent loading of the seigniorage.

Status: Fixed

Update from BetterBank: We implemeted minimal value as proposed. (f0770e8c96c959f27d1cff74b46d66812ffc8253)

7.5 [Medium] UniswapV2 price TWAP accumulator underflow will lead to oracle update failure

File(s): `LPOracle.sol`

Description: The `LPOracle` contract contains `update()` function which is used to update the TWAP price accumulator in its storage. As the accumulator returned by the UniswapV2 Pair contract only grows it is necessary to handle the underflow using the unchecked clause which is not present:

```
uint32 dt = blockTs - blockTimestampLast;
require(dt > 0, "Oracle: ZERO_TIME");

price0Average = FixedPoint.uq112x112(
    uint224((p0C - price0CumulativeLast) / dt)
);
price1Average = FixedPoint.uq112x112(
    uint224((p1C - price1CumulativeLast) / dt)
);

price0CumulativeLast = p0C;
price1CumulativeLast = p1C;
blockTimestampLast = blockTs;
```

The correct implementation is present at the `UniTWAPOracle` contract:

```
unchecked {
    // Overflow desired wrapped in unchecked
    timeElapsed = blockTimestamp - blockTimestampLast;
}
// [...]
unchecked {
    // Overflow desired wrapped in unchecked
    price0Delta = price0Cumulative - price0CumulativeLast;
    price1Delta = price1Cumulative - price1CumulativeLast;
}
```

The underflow will render the `update()` function revert and hence the `consult()` will always returned stale TWAP as there is no check against it.

The problem exists also on the USD accumulator:

```
uint256 newUsd0C = usd0CumulativeLast + u0 * dtU;
uint256 newUsd1C = usd1CumulativeLast + u1 * dtU;
```

Impact: Lack of correct up to date collateral prices combined with lack of staleness check will result in incorrect stale prices of the collateral tokens.

Recommendation(s): Add the nessary `unchecked` clause. Consider adding staleness check into the `consult` function.

Status: Fixed

Update from BetterBank: 1e7747aa89f79ff8ad55e8facc0a0e3a5cc3a699 Wrapped into unchecked

7.6 [Medium] Zapper can retain dust affecting depositor funds

File(s): `Zapper.sol`

Description: The `Zapper` contract supports "zapping" a base token into a UniswapV2 LP by swapping half of the provided base token to the second pool token and then calling `router.addLiquidity(...)`, after which LP tokens are forwarded into the lending `POOL`. The function that perform this flow include `zapToken(...)`. The result of adding liquidity to the UniswapV2 pool is that not all tokens provided could be added. Those leftovers stay on the `Zapper` contract and the `_refundDust(...)` is called to transfer them back to the caller.

The `zapToken(...)` is not the only function adding liquidity, other are:

- `addLiquidity(...);`
- `addLiquitityETH(...);`
- `requestFlashLoan(...)` - through the callback `executeOperation(...);`

The three above functions lack the `refundDust(...)`, hence leftover tokens will stay on the contract.

Impact: Users zapping through flows without `_refundDust(...)` lose residual token balances, causing silent value leakage.

Recommendation(s): Add consistent dust refunds after every code path that performs `_addLiquidity(...)`, ensuring leftover token balances are returned to the intended recipient.

Status: Fixed

Update from BetterBank: Fixed with 95d6d306b0de5aecae0d1f435421c96394c3c162

7.7 [Low] Lack of `whenNotPaused` modifier on `claimReward`

File(s): `Staking.sol`

Description: The `Staking.sol` contract allows stakers to claim their accrued rewards through the `claimReward()` function. While functions like `stake(...)`, `withdraw(...)` (which also calls `claimReward()` inside) are protected by the `whenNotPaused` modifier, the `claimReward()` isn't.

Impact: Lack of consistent pausing protection mechanism in the contract.

Recommendation(s): It is recommended to place also the modifier on the `claimReward()`. To make it work it is advised to move the claiming code into an internal `_claimReward()` function and call it within all other public functions.

Status: Fixed

Update from BetterBank: Fixed in 20c8b1fd26d655ad834738ae01af71187cce1edd

7.8 [Low] Normalization function can underflow with tokens having more than 18 decimals

File(s): `PulseMinter.sol`

Description: The helper function `_normalizeTokenAmount(...)` converts a token's amount into a standardized 18-decimals representation. It checks the token's decimals using `IERC20Metadata(token).decimals()` and, if the token uses fewer than 18 decimals, multiplies the amount by `10 ** (18 - tokenDecimals)`. However, the function assumes that `tokenDecimals <= 18`. If a token reports more than 18 decimals, the subtraction (`18 - tokenDecimals`) will underflow, causing the transaction to revert. The root cause is the absence of a defensive check on tokens with higher decimal counts.

Impact: If a token with more than 18 decimals is ever added, this function would always revert, breaking minting of Esteem flows that rely on normalization.

Recommendation(s): Update `_normalizeTokenAmount(...)` to handle tokens with more than 18 decimals.

Status: Acknowledged

Update from BetterBank:

7.9 [Info] CREATE2 vulnerability in favor token allows sell tax bypass

File(s): `Favor.sol`

Description:In the `Favor.sol` contract, the `_update()` function applies a sell tax when tokens are transferred to a contract address. This mechanism identifies contract destinations by checking if the receiving address has code deployed.

```
bool destinationIsContract = _to.code.length != 0;
// [...]
if (destinationIsContract) {
   taxAmount = (_value * sellTax) / MULTIPLIER;
}
```

However, this implementation is vulnerable to a CREATE2 bypass attack. An attacker can:

a. Pre-compute a contract address using CREATE2 opcode (without deploying any code);
b. Transfer tokens to this computed address (which passes the `code.length == 0` check and avoids the sell tax);
c. Deploy a contract to that same address using CREATE2;
d. The contract now controls tokens that were transferred without paying the intended sell tax;

Impact: This vulnerability allows users to, bypass the 50

Recommendation(s): There is no real prevention mechanism against this, yet the impact is minimal.

Status: Acknowledged

Update from BetterBank:

7.10 [Info] Protocol not prepared to handle base tokens with fee-on-transfer

File(s): `PulseMinter.sol`

Description: The protocol currently supports PLS, PLSx and pDAI as base tokens, and in the future more base tokens are planned to be supported. Currently the way how the protocol handles transfers does not support the fee-on-transfer tokens and hence all calculations which calculate Esteem or Favor token amounts accrued to users will not be valid.

Impact: Currently there is no impact. This issue is just to raise awareness that special care needs to be undertaken when introducing a new base token with fee-on-transfer feature.

Recommendation(s):If fee-on-transfer tokens are added in the future, calculate the actual received amount by checking the contract's balance before and after transfers every time the token is transferred from the user and to other accounts, like the treasury.

Status: Acknowledged

Update from BetterBank:

7.11 [Info] `UniTWAPOracle` and `LPOracle` state can become stale if it is not updated

File(s):

- `PulseMinter.sol`;
- `UniTWAPOracle.sol`;
- `LPOracle.sol`;

Description: The `PulseMinter.updateEsteemRate()` function is intended to increment `esteemRate` by `dailyRateIncrease` (0.25 ether per hour), but the increase only occurs when the function is explicitly called. If no call is made, both the rate and epoch remain unchanged.

Similarly, in `UniTWAPOracle.update()` and `LPOracle.update()`, only approved users can trigger updates to refresh cumulative price data and compute new averages for `price0Average` and `price1Average`. If these updates are not invoked, the averages remain stale and consult may return outdated pricing.

Impact: If approved users fail to trigger updates, `esteemRate` and oracle price averages may become stale, leading to outdated state or inaccurate values being returned.

Recommendation(s): To ensure `esteemRate` and oracle values remain up to date without depending solely on manual calls, implement one of the following approaches:

a. Epoch-Based Auto-Update;

Modify the functions so that whenever they are called, they first check whether the epoch time has elapsed. If so, the update logic is executed automatically before continuing with the normal function logic.

2. External Automation (Cron Job / Keeper) Integrate with an off-chain automation system (e.g., Chainlink Keepers, Gelato, or server-based cron jobs) to ensure `updateEsteemRate()` and oracle `update()` functions are called once per hour. This guarantees consistent updates without relying on users.

Status: Acknowledged

Update from BetterBank: Keeper job is planned. There is already a keeper job for seignorage allocation in place

7.12 [Best Practice] Usage of two-step ownership transfer is recommended

File(s):

- `Favor.sol`;
- `Zapper.sol`;
- `MinterOracle.sol`;
- `FavorTreasury.sol`;
- `Staking.sol`;
- `Epoch.sol`;
- `Esteem.sol`;

Description: The `Ownable2Step` pattern is an improvement over the traditional `Ownable` pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original `Ownable` pattern, where ownership can be transferred directly to a specified address, the `Ownable2Step` pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Impact: Without the `Ownable2Step` pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract.

Recommendation(s): It is recommended to use either `Ownable2Step` or `Ownable2StepUpgradeable` depending on the smart contract.

Status: Fixed

Update from BetterBank: Was implemented as proposed 9c5239aecb5d519b91df85ccccbd196fb5314f13

8 Issues - Fix Review Phase

8.1 [Medium] Flash loan functionality in `LPZapper` always reverts

File(s): `LPZapper.sol`;

Description: In the final commit, the `LPZapper` contract introduced a reentrancy guard to mitigate potential attack vectors. However, the `nonReentrant` modifier was applied to both `requestFlashLoan(...)` and `executeOperation(...)`. Since flash loans necessarily re-enter the contract through `executeOperation(...)`, this setup causes every flash loan attempt to revert, making the functionality unusable.

Impact: Flash loan functionality is permanently broken and cannot be executed.

Recommendation(s): Remove the `nonReentrant` modifier from `executeOperation(...)`.

Status: Fixed

Update from BetterBank: Resolved in f484e550e31e8d9775bfb49db423c04bb49bbb5a

9 Additional Notes

This section provides supplementary auditor observations regarding the code. These points were not identified as individual issues but serve as informative recommendations to enhance the overall quality and maintainability of the codebase.

- In `executeOperation(...)` within `Zapper.sol#109`, the addition of `+50` to the `deadline` parameter (set as `block.timestamp`) is unnecessary. The call would succeed with just `block.timestamp`.
- The calculations for `avg0Raw` and `avg1Raw` in `LPOracle.sol#173-174` can be simplified. Both expressions ultimately reduce to `avg0Raw = u0`, and the intermediate steps add unnecessary complexity.

Update from BetterBank: Fixed in f0770e8c96c959f27d1cff74b46d66812ffc8253

10 Centralisation and Trust Assumptions

While the contracts reviewed implement the intended functionality, it is important to highlight the degree of centralization present in the protocol's current design. Specifically, privileged roles controlled by the protocol team retain the ability to perform actions that can materially affect users' holdings and the overall token economy.

For example, administrators can update the list of authorized minters and, as a result, mint arbitrary amounts of new tokens at any time. This level of discretionary control introduces trust assumptions: users must rely on the protocol team to exercise these powers responsibly and in alignment with the project's stated goals.

From a security perspective, these mechanisms are not vulnerabilities in the traditional sense (e.g., coding errors or exploitable bugs), but they do represent governance and centralization risks. The current architecture provides few on-chain restrictions preventing misuse of administrative privileges.

Recommendation: The project may wish to consider measures to mitigate these risks over time, such as:

- Implementing time locks or multi-signature controls on privileged functions.
- Introducing transparent governance processes for updating critical parameters.

11 Evaluation of Provided Documentation

The **BetterBank** documentation was provided in the form of web portal where all functionalities of the protocol were documented. Additionally parts of the code contained NatSpec comments laying out the intended calculations, especially where precision was involved. The documentation could be further improved by adding:

- **NatSpec comments:** All the functions of the code included Natspec comments. Such comments coule be helpful in understanding the overall functionality of the protocol and particularly effective in areas where they explained specific design decisions.
- **Diagrams:** The documentation could be further improved by providing diagrams with token flows, especially where multiple transfers were involved in a function, like at `requestFlashLoan` in the `Zapper` contract.

The documentation provided was overally good and sufficient for the scope of the audit. The protocol could benefit from a more technically descriptive external specification to further improve clarity and developer usability. Nevertheless, the BetterBank team remained consistently available and responsive, promptly addressing all questions and concerns raised by **CODESPECT** during the audit process.

12 Test Suite Evaluation

12.1 Compilation Output

```
% npx hardhat compile
Compiling your Solidity contracts...
Downloading solc 0.8.20
Downloading solc 0.5.16
Downloading solc 0.6.6
Downloading solc 0.8.20 (WASM build)
Downloading solc 0.5.16 (WASM build)
Downloading solc 0.6.6 (WASM build)
[...]
Compiled 1 Solidity file with solc 0.5.16 (evm target: istanbul)
Compiled 1 Solidity file with solc 0.6.6 (evm target: istanbul)
Compiled 28 Solidity files with solc 0.8.20 (evm target: shanghai)
```

12.2 Tests Output

```
% npx hardhat test
Compiling your Solidity contracts...

Nothing to compile

Running Solidity tests



  0 passing

Running Mocha tests


  Esteem.sol
     deployment
       Should be able to create contract
    access control
       only owner methods
       only minter methods
    mint
       shall mint esteem, set and reset minters

  Favor.sol
     deployment
       Should be able to deploy and configure contract
       should mint initial supply to owner
    access control
       not owner shall not be able to call those methods
       not mnter  shall not be able to call those methods
       only log byuer can call this
    mint
       shall mint favor, set and reset minters
    settings
       shall change settings (39ms)
    calculations
       should calculate proper favor bonuses if Favor below 3.00 TWAP
       No bonus if TWAP above 3.00
       No bonus if TWAP == 3.00
       shall revert bonus calculation if esteem rate is 0
    transfer  and taxation
       transfer and taxation of favor tokens between end users are free
       transfer to non whitelisted contracts is taxed
       shall be able to burn
       shall be able to send token to tax-excempt contract
       shall be able to send token from tax-excempt address to on whitelisted contract
    log buy and esteem minting
       shall calculate and log proper esteem bonus for user
```

```
  FavorTreasury.sol
    deployment
       shall be able to deploy
    access control
       only owner shall be able to call those methods
    settings and initialisation
       shall initialise treasury

  LPOracle.sol
    deployment
       shall be able to deploy (60ms)
    access control
       only owner shall be able to do this (54ms)
       only approved shall be able to this (50ms)
    change settings
       shall update approval seting (50ms)
       approved user shall trigger update (55ms)

  Staking.sol
    deployment
       shall be able to deploy
    access control
       only owner shall be able to call those methods

  Zapper.sol
     deployment
       Should be able to create contract (93ms)
       Should be able to set values (94ms)
    access control
       only owner methods (99ms)
       shall no allow invocation from a wrong pool (91ms)
       shall no allow invocation from a wrong pool caller (95ms)
    token managements
       manage dust tokens (97ms)
       manage favor tokens (94ms)
       shall withdraw tokens as admin (92ms)
       shall withdraw PLS as admin (92ms)
    zapping operation
       shall not allow flash loan for unknoww tokens (89ms)
       shall request flash loan properly (101ms)
       shall refuse operation if invoked fron not registered pool (90ms)
       shall refuse operation if invoked fron wrong initiator (87ms)
       shall refuse operation  if user does not match (94ms)
       shall perform borrowing operation (108ms)
    zapping and selling
       zap  tokens  into LP (108ms)
       zap  PLS  into LP (104ms)
    buy and sell
       shall not sell unknown favor token (94ms)
0n
       shall sell token without taxes for tax exempt sellers (94ms)
       shall tax on sale if not a tax exempt seller and send + auto deposit to pool for treasury (100ms)
       shall not buy favor if not a registerred base token (90ms)
       shall buy favor, and give out bonuses to treasury and receiver (99ms)
    liquidity management
       shall refuse to add liquidity if not a registered favor (94ms)
       shall refuse to add ETH liquidity if not a a proper favor (90ms)
       shall add liquidity to registered favor (96ms)
       shall add  eth to registered favor LP (95ms)


  57 passing (3s)
```

12.3 Notes on the Test Suite

The provided test suite includes good coverage of basic and expected flows of the most important contracts, focusing primarily on common usage scenarios. The following points were identified where further improvement to the suite should be addressed:

- `FavorTreasury` - The coverage should be improved by adding pausing impact on operation. The `allocateSeigniorage` function should be thoroughly tested.

- `LPOracle` - only contains basic settings and access control tests. This should be bolstered by adding complete tests focus on the corectness of mathematical formulas.
- `UniTWAPOracle` and `MinterOracle` - should have their own set of tests focusing on formulas correctness
- `Staking` - lacks basic functionality testing. This needs to be added along with pausing impact edge cases checks.

The test suite overally shows well established testing approach, however it is incomplete and needs to be improved.

Furthermore CODESPECT advises against using mock contracts for testing integrations, if possible, as those by definition exhibit limited behavior compared to the original systems they are mimicking.