BetterBank

SMART CONTRACTS REVIEW

zokyo

July 21st 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING BETTERBANK

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 3 High issues: 3 resolved = 0 points deducted
- 9 Medium issues: 7 resolved and 2 acknowledged = - 6 points deducted
- 22 Low issues: 15 resolved and 7 acknowledged = - 7 points deducted
- 8 Informational issues: 5 resolved and 3 acknowledged = 0 points deducted

Thus, 100 - 6 - 7 = 87

TECHNICAL SUMMARY

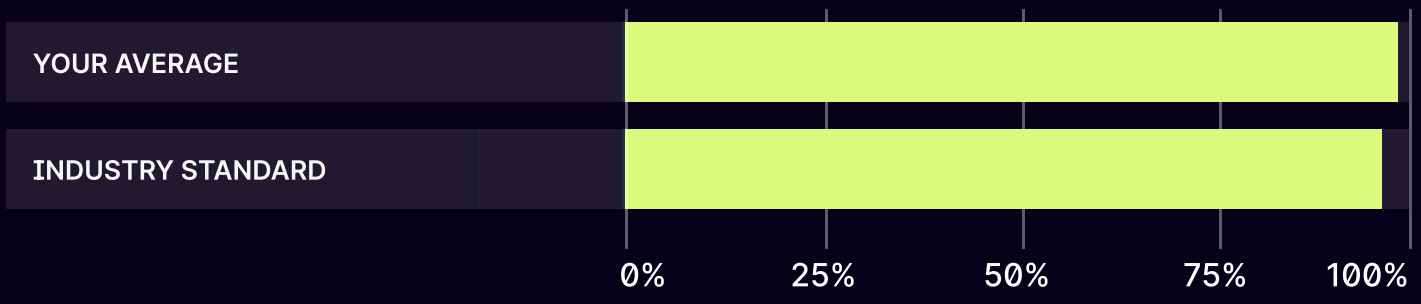
This document outlines the overall security of the BetterBank smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the BetterBank smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

Testable Code



98.32% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the BetterBank team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

Auditing Strategy and Techniques Applied 5

Executive Summary 7

Structure and Organization of the Document 9

Complete Analysis 10

Code Coverage and Test Results for all files written by Zokyo Security 52

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the BetterBank repository:
Repo: https://github.com/grape-finance/BB-Custom-contracts

Last commit - cc97d62c2afa36d4d4c7a5da4346112f43b697eb

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- Esteem.sol
- favorPLS.sol
- PulseMinter.sol
- Staking.sol
- FavorTreasury.sol
- uniswapWraper.sol
- UniTWAPOracle.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of BetterBank smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contract/s by industry leaders.
03	Testing contract/s logic against common and uncommon attack vectors.
04	Thorough manual review of the codebase line by line.

Executive Summary

The BetterBank protocol intends to revolutionize the way users lend, borrow and build wealth over time. Traditionally, banks tend to be slow, expensive and very exclusive when it comes to lending and borrowing. This is especially true across borders. BetterBank intends to bring the freedom of financial instruments to everyone regardless of origin, net worth or financial background built on transparent and immutable blockchain technology. They intend to reduce the middlemen to zero, create fairer fees and maximize yield opportunities for the users.

Zokyo was tasked with the security review of the core infrastructure which includes the following solidity contracts:

- Esteem: Acts as BetterBank share tokens making the user an investor and part owner of BetterBank which complies with the ERC20 standard.
- FavorPls: When staking Esteem tokens users generate Favor yield corresponding to the share of Esteem in the staking contract, specifically this is the Pluse Favor contract.
- PulseMinter: Allows users to redeem favor or mint esteem with certain tokens specified by the admin.
- Staking: Allows users to stake their Esteem tokens and be rewarded with Favor.
- FavorTreasury: This contract will govern the Favor token allowing the administrator to make various configurations such as specifying the staking contract, specifying oracles and allocating rewards for the staking contract.
- UniswapWrapper: Plugs into uniswap allowing users to trade their tokens for Favor tokens and qualify for rewards.
- UniTWAPOracle: A fixed window oracle that computes the average price for a given period - intended to be used by the above contract where uniswap prices are required to reduce the risk of manipulation.

Overall the contracts are well thought out to achieve the goals of the protocol however, the repository did not appear to use a framework such as foundry or hard hat in order to support engineering efforts. In addition to this, no unit or integration tests were observed by the security team. It is highly recommended that these are implemented and published before moving to deployment.

The vulnerabilities discovered by the security team ranged from Critical down to Informational. The most critical issues revolved around integration with the uniswap pools which is quite tricky to navigate for our industry as a whole especially if fresh tokens are deployed with no known chainlink oracle. The informational issues which address best engineering practices or identify attack surfaces with no security implications addresses centralisation risks and outdated coding standards in the dependency contracts.

It is recommended that the BetterBank development team carefully reviews the findings of this report and implements the suggested fixes to which a fix review will take place to ensure that the issues are resolved and that no new bugs are introduced.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the BetterBank team and the BetterBank team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

- **Critical**

  The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

- **High**

  The issue affects the ability of the contract to compile or operate in a significant way.

- **Medium**

  The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

- **Low**

  The issue has minimal impact on the contract’s ability to operate.

- **Informational**

  The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Price update can silently fail leading to the use of stale prices	High	Resolved
2	Users will lose funds if they use less amount than redeemRate when minting esteem tokens	High	Resolved
3	Epoch Start Timer Gets Overriden On Stake	High	Resolved
4	Incorrect slippage amount can lead to receiving a worse result for swapping tokens	Medium	Resolved
5	The excluding from the total supply feature can be bypassed	Medium	Acknowledged
6	Transactions from the same user within Staking.sol are getting blocked for 1 minute	Medium	Resolved
7	Incorrect use of `block.number` in L2 blockchains	Medium	Resolved
8	Incorrect Percentage Calculation If Quote Token Is Non-18 Decimal Token	Medium	Acknowledged
9	Insufficient Validation On setBonusRates In The favorPls Contract May Cause Malicious Users To Exploit Absurdly High bonusRates	Medium	Resolved
10	updateEsteemRate Can Be Sandwiched For Profit	Medium	Resolved
11	Missing Slippage While Minting Esteem With PLS/Tokens	Medium	Resolved
12	Incorrect Decimal Handling When Minting Esteem with Allowed ERC20 Tokens	Medium	Resolved

#	Title	Risk	Status
13	A Malicious User Can Utilise Flash Loans In Order To Get Free Tokens By Exploiting The Qualified Favor Bonus	Low	Acknowledged
14	The UniTWAPOracle Contract Breaks When Compiling Because Non-Abstract Contracts Cannot Have Internal Constructors	Low	Resolved
15	All Instances Of Now Have Been Deprecated Causing The Contracts To Break At Compile	Low	Resolved
16	Calls To UniswapWrapper Will Fail When USDT Is Used Because Tokens Are Not Approved To Zero	Low	Resolved
17	The User Would Need To Wait For Another Round Of Withdraw Epoch If He Claims First	Low	Resolved
18	Ether can become stuck in the contract due to a lack of a withdrawal mechanism	Low	Resolved
19	High probabilities of grove amounts being incorrectly calculated	Low	Acknowledged
20	latestAnswer Can Return Stale Prices	Low	Resolved
21	If The Token Is Fee On Transfer Token Then The Swap Would Never Work	Low	Resolved
22	Twap Function Might Return Incorrect Prices	Low	Acknowledged
23	Owner Can Assign Unlimited Minting Privileges	Low	Acknowledged
24	Unauthorized Minting Risk via Unrestricted isMinter Role Assignment	Low	Acknowledged
25	No emergency-stop on minting/seigniorage	Low	Resolved
26	Unbounded array growth invites DoS by gas exhaustion	Low	Resolved
27	Farm-and-Run Seigniorage Exploit via Immediate Stake-and-Withdraw	Low	Acknowledged

#	Title	Risk	Status
28	Insufficient Validation On _period In The Epoch Dependency Breaks >= 1 hours and <= 48 hours Invariant	Low	Resolved
29	Unhandled Error In The FavorTreasury Contract May Cause The Oracle To Report Bad Prices	Low	Resolved
30	View functions may return stale prices	Low	Acknowledged
31	Deprecated use of the `now` reserved word	Low	Resolved
32	`maxSupplyExpansionPercent` can be set lower than `minSupplyExpansionPercent` and the other way around	Low	Resolved
33	Bonus rates can be set higher than `MULTIPLIER` factor	Low	Resolved
34	Using old solidity versions can lead to multiple errors	Low	Resolved
35	Esteem and Favor PLS tokens can get minted with unlimited supply	Informational	Acknowledged
36	If `epoch` is not correctly updated within the treasury it will lead to staking features not working as expected	Informational	Acknowledged
37	Centralized behavior	Informational	Acknowledged
38	Contracts are compiled with different solidity versions	Informational	Resolved
39	Visibility in the constructor is ignored	Informational	Resolved
40	A Malicious User Can Trade Bogus Tokens To Qualify For Bonus Favor Through The UniswapWrapper	Informational	Resolved
41	Missing Future-Start Validation in FavorTreasury.initialize (FavorTreasury)	Informational	Resolved
42	Missing removeExcludedAddress in FavorTreasury	Informational	Resolved

HIGH-1 | RESOLVED

Price update can silently fail leading to the use of stale prices

Description:
The `allocateSeigniorage()` function within the `FavorTreasury.sol` smart contract executes `_updateFavorPrice()` to update favor price before using it. However, this call can silently fail:

```
function _updateFavorPrice() internal {
        try IOracle(favorOracle).update() {} catch {}
    }
```

As the dev team confirmed, the protocol may get deployed to L2 chains so the sequencer can be down.

Impact:
If the call to `.update()` fails for any reason, the transaction will not revert and a stale price will be used, leading to a wrong allocation.

Recommendation:
Implement a revert in case of not being able to update the price, in order to avoid using stale prices. It is also recommended to implement sequencer checks to ensure that it is working correctly.

Users will lose funds if they use less amount than redeemRate when minting esteem tokens

Description:
The `mintEsteemWithPLS()` and `mintEsteemWithToken()` functions within the `PulseMinter.sol` smart contract are used to mint esteem tokens in exchange for PLS and allowed tokens.

The amount of esteem tokens that the user will receive is calculated by the `_calculateEsteemMint()` function:

```
function _calculateEsteemMint(uint256 amount, address token) internal
view returns (uint256) {
        uint256 price = getLatestTokenPrice(token);
        uint256 usdAmount = (amount * price);
        return usdAmount / esteemRate;
    }
```

The key point of the issue is related to this statement: `usdAmount / esteemRate`. If users transfer an amount worth less than `esteemRate` they will receive `0` esteem tokens while their funds will get transferred anyway.

Impact:
If the described scenario takes place, users will lose their funds and will receive `0` esteem tokens.

Recommendation:
Add a require to ensure that users will receive a non `0` amount of esteem tokens, otherwise revert:

```
require(outputAmount > 0, “Increase your amount”);
```

Epoch Start Timer Gets Overridden On Stake

Description:
Consider the following scenario →

1. UserA stake X amount of tokens.
2. epochTimerStart gets set to current epoch.
3. Say Y epochs elapsed , the user would be eligible to claim rewards the next epoch , user decides to stake some more tokens and calls stake()
4. The call to stake resets the user's epochTimerStart to the current epoch , hence the user can't claim his rewards for his previous stakes for another rewardLockupEpochs.

Impact:
Users staking for the second time or more will get their previous stakes locked up again for the entire duration.

Recommendation:
Every stake should have an independent lockup period.

Incorrect slippage amount can lead to receiving a worse result for swapping tokens

Description:
The `uniswapWraper.sol` smart contract implement several different functions which interacts with uniswap's router. These functions receive different important used parameters as amountOutmin, path, etc. However, neither of these functions receive the `deadline` parameter but it is hardcoded as `block.timestamp + 900`.

This is just an example where the code shows the scenario mentioned above, but it is present across every function within the `uniswapWrapper.sol` smart contract:

```
uint[] memory amounts = uniswapRouter.swapExactETHForTokens{value:
msg.value}(
            amountOutMin, path, to, block.timestamp + 900
);
```

Impact:
Although `amountOutMin` is used to control slippage, the hardcoded deadline as `block.timestamp + 900` allows a validator (or attacker) to delay the transaction within that window. This enables them to execute the swap at the worst possible rate allowed by `amountOutMin`. For instance, if a 10% slippage is tolerated, they could ensure the transaction executes right at that 10% loss instead of, say, a 1% loss if a proper deadline were set.

Recommendation:
Set the deadline as a parameter of the functions, instead of hardcoding it to `block.timestamp + 900`.

The excluding from the total supply feature can be bypassed

Description:
The `addExcludedAddress()` function within the FavorTreasury.sol smart contract seems to be used to exclude certain amount of funds from the total supply. However, if funds are transferred to a new address, they will not be excluded.

Impact:
If it is wanted to exclude `100` tokens so that an address which holds `100` tokens is excluded and later the excluded address transfers the `100` tokens to a new non excluded one, they will be accounted again.

Recommendation:
Instead of excluding addresses from the total supply, exclude the amount of tokens.

Client comments:
This feature is only intended to be used to exclude arbitrary balances of tokens from chain burn/dead addresses or in the event tokens become stuck in contracts.

While currently we only target official burn addresses to exclude burnt tokens from supply, we might in the future use this feature for an active supply contraction mechanism

Transactions from the same user within Staking.sol are getting blocked for 1 minute

Description:
The `staking()`, `withdraw()` and `allocateSeigniorage()` functions within the `Staking.sol` smart contract implement an `onlyOneBlock` modifier:

```
modifier onlyOneBlock() {
        require(!checkSameOriginReentranted(), "ContractGuard: one block,
one function");
        require(!checkSameSenderReentranted(), "ContractGuard: one block,
one function");

        _;

        _status[block.number][tx.origin] = true;
        _status[block.number][msg.sender] = true;
    }
```

The same issue is present within `allocateSeigniorage()` function in the FavorTreasury.sol smart contract.

The `Staking.sol` smart contract uses `block.number` in different functions as an important parameter for snapshots. However, `block.number` in L2 blockchains does not work exactly as expected.

Impact:
This modifier is expected to block transactions submitted by they same user within the same block. However, the it will not work as expected but blocking transactions within the same minute (approximately). The reason is related to how `block.number` works in L2 blockchains, as described in the `Incorrect use of `block.number` in L2 blockchains` reported issue.

In Arbitrum, for example, `block.number` does not reflect the current block number of the Arbitrum blockchain but the `block.number` of the L1, which is not constantly updated over each number but once per minute.

To better understand this, look at the following example provided by the Arbitrum official documentation (https://docs.arbitrum.io/build-decentralized-apps/arbitrum-vs-ethereum/block-numbers-and-time#example)

Wall clock time	12:00 am	12:00:15 am	12:00:30 am	12:00:45 am	12:01 am	12:01:15 am
Ethereum `block.number`	1000	1001	1002	1003	1004	1005
Chain's `block.number` *	1000	1000	1000	1000	1004	1004
Chain's block number (from RPCs) **	370000	370005	370006	370008	370012	370015

Recommendation:
Consider only using nonReentrant modifier in order to block reentrancy within this functions or used a custom modifier to block transactions using block.timestamp instead of block.number or ArbSys(100).arbBlockNumber().

Consider creating different contracts for L1 and L2 deployments taking into consideration the described issue.

For extra reference: https://solodit.cyfrin.io/issues/incorrect-use-of-l1-blocknumber-on-arbitrum-cantina-none-uniswap-pdf

Incorrect use of `block.number` in L2 blockchains

Description:
The `Staking.sol` smart contract uses `block.number` in different functions as an important parameter for snapshots. However, `block.number` in L2 blockchains does not work exactly as expected.

Impact:
In Arbitrum, for example, `block.number` does not reflect the current block number of the Arbitrum blockchain but the `block.number` of the L1, which is not constantly updated over each number but once per minute

To better understand this, look at the following example provided by the Arbitrum official documentation (https://docs.arbitrum.io/build-decentralized-apps/arbitrum-vs-ethereum/block-numbers-and-time#example)

Wall clock time	12:00 am	12:00:15 am	12:00:30 am	12:00:45 am	12:01 am	12:01:15 am
Ethereum `block.number`	1000	1001	1002	1003	1004	1005
Chain's `block.number` *	1000	1000	1000	1000	1004	1004
Chain's block number (from RPCs) **	370000	370005	370006	370008	370012	370015

Recommendation:
Consider using `block.timestamp` instead of `block.number` or `ArbSys(100).arbBlockNumber()`.

Consider creating different contracts for L1 and L2 deployments, taking into consideration the described issue.

For extra reference: https://solodit.cyfrin.io/issues/incorrect-use-of-l1-blocknumber-on-arbitrum-cantina-none-uniswap-pdf

Incorrect Percentage Calculation If Quote Token Is Non-18 Decimal Token

Description:
In the FavorTreasury while allocating seigniorage the percentage of price is compared with the minSupplyExpansionPercentage and the maxSupplyExpansionPercentage both of which are expected to be basis points of the percentage (after being multiplied by 1e13) , but if the quote token in the oracle is a non-18 decimal token like USDC/USDT then the price of the favor token (getFavorPrice()) would be in USDC/USDT i.e 6 decimals and the comparison with min and max would almost always result in percentage being less than min.

Impact:
In the FavorTreasury while allocating seigniorage the percentage of price is compared with the minSupplyExpansionPercentage and the maxSupplyExpansionPercentage both of which are expected to be basis points of the percentage (after being multiplied by 1e13) , but if the quote token in the oracle is a non-18 decimal token like USDC/USDT then the price of the favor token (getFavorPrice()) would be in USDC/USDT i.e 6 decimals and the comparison with min and max would almost always result in percentage being less than min.

Recommendation:
Take into account the decimal of the token while calculating the percentage.

Client comments:
In FavorTreasury getFavorPrice() is only ever used to quote the TWAP of Favor tokens which are always deployed with standard 18 decimals.

Insufficient Validation On setBonusRates In The favorPls Contract May Cause Malicious Users To Exploit Absurdly High bonusRates

Description:
Bonus rates are used to determine how much favor users can qualify for simply for making trades. Currently by default the bonusRate is `5_000` (`50%`). There is insufficient validation when attempting to set the new bonus rates which may give an attacker an opening for exploitation.

Impact:
Accidentally setting an absurdly high bonus rate may allow an attacker to make a profit or drain the uniswap pool using the excess favor tokens or bonus set accidentally to using 1e18 decimals will cause a precision loss awarding users an absurdly high amount. This was rated a Medium in severity because this relies on external requirements but results in a significant amount of economical damage.

Recommendation:
It is recommended that the new bonus rates are validated when set so that they are of a reasonable value.

updateEsteemRate Can Be Sandwiched For Profit

Description:
In the PulseMinter contract the esteemRate increases daily (0.25 USD increase) when the keeper calls updateEsteemRate(). A user can take advantage of this step-wise jump in redeem rate by firstly minting themselves Esteem tokens using mintEsteemWithPLS (or mintEsteemWithToken) done through frontrunning keeper's call to updateEsteemRate() and then backrunning updateEsteemRate() to call redeemFavor() , since the minted tokens amount is inversely proportional to the esteem rate (inside _calculateEsteemMint()) the amount of esteem tokens minted would be higher pre-update of the rate and then user redeem if after the esteem rate increases suddenly and during redemptions the favor token mint amount is directly proportional to the esteem rate (_calculateRedeemAmount()) . Therefore, the user makes a little profit sandwiching daily esteem rate updates.

Impact:
Users can sandwich the updation of the esteem rate and gain profit instantaneously.

Recommendation:
Have a minimum lock duration for the user within which redemptions are not allowed.

Missing Slippage While Minting Esteem With PLS/Tokens

Description:
When minting Esteem tokens using mintEsteemWithPLS or mintEsteemWithToken the amount of Esteem to be minted is dependent on the esteemRate , it is possible that the current rate is too high for the user or the mint tx might execute in future (from the mempool) when the esteem rate is higher.

Impact:
Users would experience slippage due to linearly increasing esteem rate.

Recommendation:
Introduce a deadline or slippage in the mint functions.

Incorrect Decimal Handling When Minting Esteem with Allowed ERC20 Tokens

Description:
The MintRedeemer contract allows users to mint Esteem tokens using specific ERC20 tokens (allowed tokens). However, the contract incorrectly assumes all tokens have 18 decimals, which is not always true.

Impact:
Tokens like USDT have 6 decimals and failing to properly convert these decimals before performing calculations can result in significant inaccuracies in minted Esteem amounts.

Proof of Concept:
Consider the following reproducible scenario:
The owner adds USDT (which has 6 decimals) as an allowed minting token using the function:

```
MintRedeemer:setAllowedMintToken(address token, bool allowed)
```

A user attempts to mint Esteem tokens using 1,000 USDT (1,000 units, represented as `1000 * 1e6`).
The oracle returns a price of exactly $1 for USDT, represented as `1e18` (18 decimals).
The current implementation of the calculation within the following functions:

```
MintRedeemer:mintEsteemWithToken(uint256 amount, address token)
    → MintRedeemer:_calculateEsteemMint(amount, token)
        → MintRedeemer:getLatestTokenPrice(token)
```

The calculation in `_calculateEsteemMint is currently:`

```
uint256 price = getLatestTokenPrice(token);
uint256 usdAmount = (amount * price);
return usdAmount / esteemRate;
```

Plugging the values for 1,000 USDT (6 decimals) gives:

- `usdAmount = 1000e6 * 1e18 = 1000e24`
- Esteem minted: `1000e24 / 16e18 = 62.5e6`

Because Esteem is an 18-decimal token, the correct Esteem amount should be represented with 18 decimals (62.5e18), not 6 decimals (62.5e6). This error causes incorrect and unexpected token minting outcomes, either severely disadvantageing the user or unintentionally benefiting them, depending on circumstances.

Functions Involved:

- MintRedeemer:setAllowedMintToken(address token, bool allowed)
- MintRedeemer:mintEsteemWithToken(uint256 amount, address token)
- MintRedeemer:_calculateEsteemMint(uint256 amount, address token)
- MintRedeemer:getLatestTokenPrice(address token)

Recommendation:

Implement explicit token decimal handling when using ERC20 tokens for minting Esteem. Normalize token decimals to a standard 18-decimal format before calculations.

A Malicious User Can Utilise Flash Loans In Order To Get Free Tokens By Exploiting The Qualified Favor Bonus

Description:
The UniswapWrapper allows users to trade their tokens in order to receive Favor and vice versa except trading the opposite way does not qualify for bonuses. A malicious user can take out a large flash loan, trade for favor and favor bonus (tanking the favor price) then trade back to the other token. Because the price is significantly lower than previously, the bonus qualified and obtained now has more impact when trading back allowing a malicious user to extract value from the uniswap pools through the favor bonus feature.

Proof of Concept
The proof of concept below shows a user taking out a large flash loan, makes a trade, then trades back resulting in the extraction of value from the uniswap pool using qualified bonuses:
https://gist.github.com/chris-zokyo/8951f4832b975bdc494a9990baa84763

Impact:
A malicious user with a flashloan or a whale can extract value from the uniswap pools through the wrapper by exploiting the bonuses feature of the FavorPls contract.

Recommendation:
It's recommended that users need to wait a certain amount of time before being able to claim bonuses if they are to be awarded during trades.

Client comment:
Favor have a 50% sell tax, all buys through the router wrapper are given 44% bonus in Esteem (50% in original favor contract*, now updated to 44%) which has no liquidity and can only be redeemed for 70% of its value in Favor. For example, not accounting for slippage, user flashloans $1m buys favor, receives $440k bonus in Esteem token, redeems this Esteem token to Favor for a value of $308k then sells all the Favor (worth 1,308k including the bonus) with a 50% sell tax for $654k and is not able to pay back the flashloan. Even when not selling, but instead borrowing against the Favor, the yield on 1,308k Favor is $784,800k, which is also not enough to pay back the flashloan. *Even in the old setting, the yield sold/borrowed would be $675,000 / $810,000, which is also far removed from the $1.5M required to pay back the flash loan.

LOW-2 | RESOLVED

The UniTWAPOracle Contract Breaks When Compiling Because Non-Abstract Contracts Cannot Have Internal Constructors

Description:
The UniTWAPOracle relies on the Operator dependency in order to make up the Oracle contract. The contract fails to compile because of an internal constructor.

Impact:
The contract will break and fail to compile unless this issue is rectified. This was a medium in severity because the contract breaks in a way which was not expected.

Recommendation:
Remove `internal` from the Operator contract constructor.

LOW-3 | RESOLVED

All Instances Of Now Have Been Deprecated Causing The Contracts To Break At Compile

Description:
The UniTWAPOracle's dependency Epoch to check timestamps uses the now keyword which has been deprecated.

Impact:
Continued usage of `now` will cause the contracts to break on compile.

Recommendation:
Replace `now` with block.timestamp.

LOW-4 | RESOLVED

Calls To UniswapWrapper Will Fail When USDT Is Used Because Tokens Are Not Approved To Zero

Description:
The Tether USDT contract does not implement the standard approval in accordance with the ERC20 standard. In the current implementation, for each call to `approve`, the contract will revert as there is not an approval of zero.

Impact:
Users will not be able to use USDT.

Recommendation:
It's recommended that `forceApprove` is used as apart of the SafeERC20 library which initially sets an approval of zero.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol#L101-L108

LOW-5 | RESOLVED

The User Would Need To Wait For Another Round Of Withdraw Epoch If He Claims First

Description:
When a user claims his reward after staking the epochTimerStart is reset , this means that if the user had to wait rewardLockupEpochs to claim , then he claims , and then after claiming he wants to exit with withdraw() , he can not do that for another withdrawLockupEpochs since the epochTimerStart is reset while claiming. It is expected from the user to not call claim prior to withdraw since claimReward is called within withdraw() but if user decides to call claimRewards before (maybe some blocks before deciding to exit) , then the user would not be able to withdraw for another withdrawLockupEpoch.

Impact:
Users claiming before exiting would be locked for another withdrawLockupEpoch.

Recommendation:
Don't reset the user's epochTimerStart when they call claimReward; only update it on stake so that claiming rewards doesn't extend the withdraw lockup.

Ether can become stuck in the contract due to a lack of a withdrawal mechanism

Description:
The `uniswapWrapper.sol` smart contract includes a `receive()` function, allowing it to accept and hold Ether. However, there is no function implemented to withdraw that Ether. In particular, in the function `swapExactFavorForETH`, if the caller sets the `to` parameter to the contract's own address (i.e., `uniswapWrapper.sol`), the Ether from the swap will remain stuck in the contract, since no mechanism exists to recover it:

```
IUniswapV2RouterSupportingFeeOnTransfer(address(uniswapRouter))
    .swapExactTokensForETHSupportingFeeOnTransferTokens(
        amountIn,
        amountOutMin,
        path,
        to, // can be set to the contract address
        block.timestamp + 900
    );
```

Additionally, Ether can become stuck if sent directly to the contract (e.g., via .send, .transfer, or .call{value: ...}), even outside the context of a swap.

Impact:
If a user or external contract mistakenly (or even intentionally) sets the to address to `uniswapWrapper.sol` when calling `swapExactFavorForETH,` or sends Ether directly to the contract, those funds become irretrievable due to the absence of any withdrawal or rescue mechanism. This could result in permanent loss of funds.

Recommendation:
Implement an `adminWithdraw()` function restricted to the contract owner or a designated admin. This function should allow manual recovery of stuck Ether.

High probabilities of grove amounts being incorrectly calculated

Description:
The `allocateSeigniorage()` function within the `FavorTreasury.sol` smart contract executes the following operation to calculate the amount of funds which are sent to the `grove`:

`uint256 _savedForGrove = (favorSupply * _percentage) / 1e18 / 24; // divide by 24 for each epoch per day`

There is a division by 24 expecting each day having increased 24 epochs. However, epochs are only increased if `allocateSeigniorage()` is executed.

Impact:
If the function `allocateSeigniorage()` is not executed exactly 1 time each hour in a really precisely way, there will not be an increase of 24 epochs in 24 hours, so the assumption will not be correct.

Recommendation:
Refactor how the operation is performed.

Client comments:
The AllocateSeigniorage() function is called via keeper bot running a cron job on the hour every hour, expansion is assumed to increase per epoch rather than exactly per hour, so this is within assumptions. If the keeper bot fails, this function can be called by any EOA on-chain. We will implement a button in the UI so that if ever the bot fails to call, then after the hour has passed, any user can call it without having to go into the contract.

LOW-8 | RESOLVED

latestAnswer Can Return Stale Prices

Description:
Though out of scope , the Minter contract (L886) makes use of the latestAnswer function from Chainlink price feed to fetch the price of ETH (native token) , this price returned by latestAnswer() does not provide protection from stale values and is even marked as deprecated by chainlink.

Impact:
The price of the native returned by latestAnswer can be stale.

Recommendation:
Use latestRoundData instead.

LOW-9 | RESOLVED

If The Token Is Fee On Transfer Token Then The Swap Would Never Work

Description:
When swapping a token to the favor token →

```
external {
        address finalToken = path[path.length - 1];
        require(isFavorToken[finalToken], "Path must end in registered
FAVOR");

        IERC20(path[0]).transferFrom(msg.sender, address(this), amountIn);
        IERC20(path[0]).approve(address(uniswapRouter), amountIn);

        uint[] memory amounts = uniswapRouter.swapExactTokensForTokens(
            amountIn, amountOutMin, path, to, block.timestamp + 900
        );
```

If the input token is a FoT then the actual transferred amount would be lesser than amountIn and the swap would never execute.

Impact:
Swap would not work (from FoT to favor) if the token is FoT.

Recommendation:
Track the exact amount of transferred tokens and pass that amount in the uniswap's swap function.

Twap Function Might Return Incorrect Prices

Description:
In FavorTreasury the getFavorUpdatedPrice can be used to get the price of one favor token , if the timeElapsed is small (the difference between current timestamp and the latest timestamp when an update was made) then the twap price returned is more probable to be incorrect and prone to manipulation.

Impact:
twap() function might return stale/manipulated price.

Recommendation:
It should be acknowledged to the users using this functionality offchain that this price might be incorrect.

Client comment:
The TWAP function is not used in the contract and is only referenced in views for UI. A comment was added to the contracts referencing it and will also be made on the UI, where referenced, to ensure users are aware of the more variable nature of its price calculations.

Owner Can Assign Unlimited Minting Privileges

Description:
The Esteem ERC20 contract allows the owner to grant or revoke minting privileges through the `addMinter` and `removeMinter` functions. Minting is strictly controlled by addresses set as minters. However, there's no limitation, timelock, or governance process on granting minter status. The owner can assign minting rights arbitrarily and instantly to any address.

Impact:
This presents significant centralization risk: a compromised owner account or malicious behavior by the owner could grant minting privileges to a malicious actor or untrusted contract, enabling unlimited minting and dilution of token supply.

Proof of Concept:
- The owner's wallet or key management is compromised.
- The attacker (or malicious owner) calls `addMinter(attackerAddress)` to grant themselves minter status.
- The attacker then calls `mint()` repeatedly, creating an unlimited amount of Esteem tokens.
- The newly minted tokens are dumped onto the market, rapidly reducing the token's price and trust, and allowing the attacker to profit significantly.

Recommendation:
Restrict minting privileges only to trusted, audited contracts or a multi-sig controlled minter management process.

Implement a two-step ownership transfer
Add a timelock mechanism or DAO governance vote for adding or removing minters, ensuring the community can detect and prevent unauthorized minting before it happens.
Consider implementing a hard-coded maximum supply cap to prevent unlimited minting, providing an additional safeguard against severe token inflation.
Implement monitoring on-chain events (`MinterAdded`, `MinterRemoved`, and `Mint`) so suspicious or unusual activity is flagged quickly.

Client comments:
The suggestion of a multisig ownership as well as a timelock mechanism will be implemented. On top of that, off-chain security measures will be taken to protect the controlling wallets of that multisig. A hard-coded maximum supply cap would fundamentally invalidate the intended design.

LOW-12 | ACKNOWLEDGED

Unauthorized Minting Risk via Unrestricted `isMinter` Role Assignment

Description:
The FavorPLS contract allows the contract owner to grant minting privileges to any address through the `addMinter(address account)` function without additional security controls.

Impact:
This unrestricted ability to assign the minter role introduces a significant centralization risk, potentially enabling malicious or compromised addresses to mint an unlimited number of FavorPLS tokens, diluting token value and harming legitimate token holders.

Proof of Concept:
1. The contract owner calls `FavorPLS:addMinter(maliciousAddress).`
2. The malicious address gains unrestricted minting capabilities.
3. The attacker calls `FavorPLS:mint(recipient, amount)` repeatedly, creating tokens at will.

Recommendation:
Implement additional access controls and security checks, such as a multi-signature mechanism, timelock, or community governance approval before assigning minting privileges. This ensures no single address or individual can unilaterally gain control of minting capabilities, mitigating centralization and unauthorized minting risks.

Client comments:
The suggestion of a multisig ownership as well as a timelock mechanism will be implemented. On top of that, off-chain security measures will be taken to protect the controlling wallets of that multisig. A hard-coded maximum supply cap would fundamentally invalidate the intended design.

No emergency-stop on minting/seigniorage

Description:
There is no “pause” or “circuit breaker” on `allocateSeigniorage`, `stake`, or `withdraw`. If something goes wrong—an oracle glitch, a flash-loan price pump, or any other bug—the team cannot halt new minting or deposits.

Impact:
Without a pause mechanism, an oracle manipulation or unexpected bug could trigger uncontrolled seigniorage minting or corrupted staking/withdraw flows. The protocol would continue to mint or accept deposits despite being exploited, enabling an attacker to:

- **Drain the treasury** by repeatedly calling `allocateSeigniorage` under manipulated prices.
- **Inflate the token supply** without check, diluting every holder’s share.
- **Lock or steal user funds** if stake/withdraw logic is broken, with no way to halt deposits or withdrawals mid-exploit.

Contract & Function(s)

- **FavorTreasury.allocateSeigniorage**
- **Staking.stake, Staking.withdraw**

Proof of Concept
An attacker manipulates the TWAP oracle or triggers a bug and begins draining or over-minting protocol tokens. Without a `whenPaused` guard, you cannot pause the contract to stop further damage.

Recommendation:
apply `whenNotPaused` to all mutative entry points. Give the owner (or a multisig) the ability to pause/unpause the protocol instantly if anything suspicious occurs.

Unbounded array growth invites DoS by gas exhaustion

Description:
Both the treasury's `excludedFromTotalSupply[]` and the staking contract's `groveHistory[]` grow without bound. Every call to `getFavorCirculatingSupply()` or history-based views loops over the entire array, eventually running out of gas.

Impact:
if the owner (or governance) inadvertently adds many entries, routine reads (e.g. `canWithdraw(), rewardPerShare()`, or a frontend polling `getFavorCirculatingSupply()`) will run out of gas and revert. That halts staking, withdrawals, reward claims, and seigniorage allocation—denying service to everyone. Even "view" functions cost gas when invoked internally by non-view code, so nothing in the protocol can safely touch those arrays once they get too big.

Contract & Function(s)
- **FavorTreasury.getFavorCirculatingSupply** (loops `excludedFromTotalSupply`)
- **Staking.latestSnapshotIndex / GroveSnapshot[]**

Proof of Concept
The contract owner (or governance) may accidentally add dozens or hundreds of entries. Soon, even simple reads (`canWithdraw, getFavorPrice`, or UI calls to `rewardPerShare`) run out of gas and revert, effectively brick-walling the protocol.

Recommendation:
- Cap the maximum length of these arrays, or
- Migrate to a ring-buffer / sliding window for snapshots, and
- Store a running total for excluded balances instead of looping every time.

Farm-and-Run Seigniorage Exploit via Immediate Stake-and-Withdraw

Description:
An attacker can game the seigniorage distribution by staking a minimal amount of tokens immediately before the treasury's `allocateSeigniorage()` call, collecting their pro-rata share of freshly minted rewards, and then withdrawing their stake in the same epoch. Because there is no lockup or snapshot window, they can repeat this every epoch at near-zero cost

Impact:
The protocol's reward mechanism can be drained by "flash" stakes of arbitrarily small size. An attacker staking 10 FAVOR into a 10 000 FAVOR pool could mint ≈1 FAVOR per epoch for free, repeatedly, diluting legitimate stakers and draining seigniorage over time.

Recommendation:
Disallow withdrawal or reward claims for stakes made within the current epoch.

Client comments:
Favor is not staked in the Groves. Esteem is staked in the Groves. Since AllocateSeigniorage() is called for allocation of all Favor types in one block, there is no possibility for Esteem stakers to move their Esteem to another Favor Grove in time to receive that Grove's Favor as well, which if it were possible would be the only benefit to farm-and-run. Since Esteem cannot be sold at full value, anyone trying to buy-farm-sell would incur serious losses. If users wish to spend the gas for staking and unstaking, they're welcome to do so as often as they want between the calls for AllocateSeigniorage().

It is true that if users would suddenly flood an otherwise sparsely staked grove with Esteem right before AllocateSeigniorage(), the pool allocation per Esteem would be diluted for everyone. This is however an intended part of the strategy game that is the Wildlands.

Insufficient Validation On _period In The Epoch Dependency Breaks >= 1 hours and <= 48 hours Invariant

Description:
The `setPeriod` function validates that period is at least 1 hour and less than 48 hours before setting a new period; however, this is not done in the contract constructor which breaks this invariant.

Impact:
The period can be set to less than an hour or more than 48 hours in the contract constructor which breaks an invariant. This was rated low in severity because this is only possible in the contract constructor.

Recommendation:
Validate period in the constructor similarly to the validation in `setPeriod` (at least more than 1 hour but less than 48 hours).

Unhandled Error In The FavorTreasury Contract May Cause The Oracle To Report Bad Prices

Description:
The `_updateFavorPrice` in the FavorTreasury contract makes a call to `update()` which updates the 1 day EMA price from Uniswap allowing an accurate reading of consult and twap. A try catch block is used when updating prices which will pass even if the call to `update()` fails.

Impact:
If a call to `update()` fails for some reason, nothing is done in the catch block which might prompt the uniswap oracle to return bad prices when allocating seigniorage. This was a medium in severity because this relies on edgecase scenarios to be met.

Recommendation:
It is recommended that the try catch block is removed or raise an exception through a require statement if the call to update fails which ensures that accurate prices are always reported.

LOW-18 | ACKNOWLEDGED

View functions may return stale prices

Description:
There are several view functions which call `.consult()` without first calling update(), for example: `getFavorPrice()`.

Impact:
If `update()` is not called, they may return a stale price.

Recommendation:
Call `update()` before calling `consult()`.

Client comments:
In FavorTreasury the favor oracle is updated every hour during the AllocateSeigniorage call, after this update getFavorPrice() view is used to determine the current TWAP price for printing. The oracle utilizes this 1hour period to compute the price average which is important for all minting and redeeming functions on all contracts however should only be updated once per hour to compute an average over that time and not on a shorter time period. Updating the oracle every hour using keepers ensures that the price average remains fresh and with a sufficient TWAP for all functions that use consult().

LOW-19 | RESOLVED

Deprecated use of the `now` reserved word

Description:
The `Epoch` smart contract within the `UniTWAPOracle.sol` file is using the `now` solidity reserved word which is deprecated.

Impact:
Using deprecated words can lead to unexpected behaviors.

Recommendation:
Use the current version of the contract and use `block.timestamp` instead of `now`.

LOW-20 | RESOLVED

`maxSupplyExpansionPercent` can be set lower than `minSupplyExpansionPercent` and the other way around

Description:
`maxSupplyExpansionPercent` can be set lower than `minSupplyExpansionPercent` and the other way around within the `FavorTreasury.sol` smart contract.

Impact:
If these variables are selected wrong, they will have a negative impact within the calculations in `allocateSeigniorage()`, leading to an incorrect amount of funds saved for the grove.

Recommendation:
Introduce some checks to ensure that these variables are placed correctly.

LOW-21 | RESOLVED

Bonus rates can be set higher than `MULTIPLIER` factor

Description:
The `setBonusRates()` function within the `favorPLS.sol` smart contract is used to set new values for `bonusRate` and `treasuryBonusRate`. However, there is not a higher maximum value check for these new values.

The same situation applies to the `setTreasuryBonus()` function within the `PulseMinter.sol` smart contract: `treasuryBonusRate` can be set higher than `MULTIPLIER`

Impact:
If bonus rates values are set higher than `MULTIPLIER` then the operations will grow higher than 100%, which is something unexpected.

Recommendation:
Add 2 require statements to ensure that `_bonusRate` and `_treasuryBonusRate` are lower or equal than `MULTIPLIER` or any other selected maximum threshold.

Using old solidity versions can lead to multiple errors

Description:
The `UniTWAPOracle.sol` smart contract is using old solidity versions:

`pragma solidity >=0.6.0 <0.8.0;`

Impact:
Using old solidity versions can lead to multiple errors like uncontrolled overflow/underflow.

Recommendation:
Use solidity versions higher than 0.8.0.

Esteem and Favor PLS tokens can get minted with unlimited supply

Description
The `Esteem.sol` smart contract implements a `mint()` function which allows minters to mint tokens without a limited supply.

```
function mint(address recipient_, uint256 amount_) public onlyMinter {
        _mint(recipient_, amount_);
    }
```

The same situation applies to the `favorPLS.sol` smart contract:

```
function mint(address recipient_, uint256 amount_) external {
        require(isMinter[msg.sender], "Not authorized to mint");
        _mint(recipient_, amount_);
     }
```

Client comments:
We don't intend to have supply limits on Favor or Esteem tokens as the system is designed to expand perpetually, and faster or slower based on demand. While Esteem has a contractive function through the smelter, Favor does not and this is intentional. In times of low demand, the low price of Favor will create effective value contraction without supply contraction. Since Favor is primarily a collateral asset, this is enough.

If `epoch` is not correctly updated within the treasury it will lead to staking features not working as expected

Description
The main functions within the `Staking.sol` smart contract relies on the `epoch` variable from the `Treasury.sol` smart contract. If the variable is not updated correctly within the treasury, the functionalities in Staking will not work as expected.

Recommendation:
Make sure to call and often update and correctly the `epoch` variable within the treasury contract.

Client comments:
We use a keeper bot to call function AllocateSeigniorage() on the FavorTreasury contract each hour which will update the TWAP price of the Favor token in the Oracle contract and also update the epoch. If the keeper bot fails, this function can be called by any EOA on-chain. We will implement a button in the UI so that if ever the bot fails to call, then after the hour has passed, any user can call it without having to go into the contract.

INFORMATIONAL-3 | ACKNOWLEDGED

Centralized behavior

Description

- `logBuy()`: this function which can only be called by `isBuyWrapper` users can be called at any time, any times as wanted and the `amount` parameter can be set to any value, leading to the inflation of `pendingBonus` for `user` and the mint of esteem tokens for the treasury.
- `PulseMinter.sol`: admin can withdraw any amount of tokens and ether from the contract.

Client comments:
logBuy() - We want to reward any user buy of Favor with Esteem as long as the Favor has a TWAP price of lower than 3.5x its paired asset, and doing so through the wrapper allows us to control this best. We have no intention on limiting Esteem minting bonus/supply for users. PulseMinter.sol - We have implemented an admin withdraw function for tokens or PLS in the contract only for purposes of recovering stuck tokens mistakenly sent to the contract. Essentially this contract holds no tokens for users. All tokens are immediately transferred in and out instantly on all functions, so the only time a token would be left in the contract is from a mistaken transfer by a user, in which case these functions allow us to recover the tokens for users.

INFORMATIONAL-4 | RESOLVED

Contracts are compiled with different solidity versions

Description
Several smart contracts are compiled with different solidity versions which is not recommended.

Recommendation:
Use the same solidity version for every smart contract as it is recommended as a good practice.

INFORMATIONAL-5 | RESOLVED

Visibility in the constructor is ignored

Description
The constructor within the `UniTWAPOracle.sol` smart contract is marked as public, however visibility in constructors are ignored.

Recommendation:
Remove the `public` word from the constructor.

INFORMATIONAL-6 | RESOLVED

A Malicious User Can Trade Bogus Tokens To Qualify For Bonus Favor Through The UniswapWrapper

Description
The UniswapWrapper allows users to specify a path in which users can trade their tokens then qualify for their bonus, for example `swapExactTokensForFavorAndTrackBonus`. This function will log a buy with the FavorPls contract and track their bonus. A user can supply liquidity using a connecting token and a bogus token then make trades to qualify for bonuses using .

Recommendation:
It's recommended that bonuses are only qualified for specific tokens, this can be done by validating that the path is of length 2, if not, the trade does not qualify for bonuses.

Missing Future-Start Validation in FavorTreasury.initialize (FavorTreasury)

Description

The `FavorTreasury.initialize()` Contract accepts any `_startTime`, including values in the past or zero, without check. If the DAO accidentally sets it too far in the past, epochs begin immediately (or if zero, `nextEpochPoint() == 0` and expansion can be triggered continuously).

Recommendation:

Enforce that the treasury launch is scheduled strictly in the future by adding a check at the top of `initialize(...)`

```
require(_startTime > block.timestamp, "Must start in future");
```

This ensures that the DAO must deliberately choose a future timestamp, preventing accidental immediate activation or zero-time scheduling.

Missing `removeExcludedAddress` in FavorTreasury

Description

In the `FavorTreasury` contract's supply-exclusion mechanism, the owner can call `addExcludedAddress(address)` to omit certain addresses from the circulating-supply calculation—but there is no complementary `removeExcludedAddress(address)`. Once an address is excluded, it remains excluded forever, even if that was a mistake or conditions

Recommendation:

Add a `removeExcludedAddress(address)` function that:

1. Checks `require(_isExcluded[addr], "not excluded");`
2. Marks `_isExcluded[addr] = false;`
3. Removes `addr` from the `excludedFromTotalSupply` array

	Esteem.sol favorPLS.sol PulseMinter.sol Staking.sol FavorTreasury.sol uniswapWraper.sol UniTWAPOracle.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting BetterBank in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the BetterBank contract/s requirements for details about issuance amounts and how the system handles these.

Esteem

✓

✓
✓
✓
✓
✓
✓

FavorPLS

✓
✓

✓
✓
✓
✓
✓
✓

✓
✓
✓
✓
✓
✓
✓
✓
✓

Claim

✓ Should claim bonus (361ms)
✓ Should have available bonus

Transfer with Tax

✓ Should pay tax (1489ms)
✓ Tax exempt (recipient) should not pay tax (77ms)
✓ Tax exempt (sender) should not pay tax (115ms)
✓ Non market pair should not pay tax (282ms)

Set

✓ Should set treasury
✓ Treasury should not be zero
✓ Only admin should set treasury
✓ Should set esteem
✓ Esteem should not be zero
✓ Only admin should set esteem
✓ Should set sell tax
✓ Tax should not exceed max value
✓ Only admin should set sell tax
✓ Should set bonus rates
✓ Only admin should set bonus rates
✓ Should set tax exempt
✓ Only admin should set tax exempt
✓ Should set pair
✓ Only admin should set pair

FavorTreasury

Deployment

✓ Should set the right variables (5500ms)
✓ Should initialize only once
✓ Should initialize with valid values (442ms)

Set

✓ Should set a new grove
✓ Only owner should set a new grove
✓ New grove should not be zero
✓ Should set a new oracle
✓ Only owner should set a new oracle
✓ New oracle should not be zero
✓ Should set a max percentage
✓ Only owner should set a max percentage
✓ Should set a valid value for max percentage (42ms)
✓ Should set a min percentage
✓ Only owner should set a min percentage
✓ Should set a valid value for min percentage

✓ Should add excluded address (459ms)
✓ Only owner should add excluded address
✓ Should add extra fund (477ms)
✓ Only owner should add extra fund
✓ Should set a valid value for extra fund

allocateSeigniorage

✓ allocateSeigniorage (5910ms)
✓ Invalid favor price (163ms)
✓ allocateSeigniorage (percentage > max) (312ms)
✓ allocateSeigniorage (percentage < min) (241ms)
✓ Only one block (528ms)
✓ Should revert before start time (332ms)
✓ Should revert before next epoch (462ms)

Governance Recover

✓ Should recover (273ms)
✓ Only owner should recover
✓ Should not recover favor

Get twap price

✓ Should get twap price (38ms)
✓ Should revert when getting an error from oracle

PulseMinter

Deployment

✓ Should set the right variables (2954ms)

Mint Esteem With PLS

✓ Should mint esteem with pls (1183ms)
✓ Should send valid value
✓ Should mint when only unpaused

Mint Esteem With Token

✓ Should mint esteem with token (1752ms)
✓ Should send valid value
✓ Token should be accepted
✓ Should mint when only unpaused

Redeem favor

✓ Should redeem (1256ms)
✓ Should only redeem supported favor token
✓ Should only redeem valid amount
✓ Should redeem when only valid favor price (42ms)
✓ Should redeem when only valid oracle
✓ Should only redeem when not paused (38ms)

Set

✓ Should set daily rate increase

✓ Only owner should set daily rate increase
✓ Should update esteem rate (54ms)
✓ Only once per a day
✓ Only keeper should update rate
✓ Should set redeem rate
✓ Only owner should set redeem rate
✓ New rate should not exceed max
✓ Should set esteem rate
✓ Only owner should set esteem rate
✓ Esteem rate should not be zero
✓ Should set treasury bonus
✓ Only owner should set treasury bonus
✓ Should set treasury
✓ Only owner should set treasury
✓ Treasury should not be zero
✓ Should set active favor token (458ms)
✓ Only owner should set active favor token
✓ Favor token should not be zero
✓ Should set keeper
✓ Only owner should set keeper
✓ Only owner should set oracle
✓ Oracle should not be zero
✓ Only owner should allow tokens
✓ Zero address should not be allowed
✓ Owner should pause/unpause (41ms)
✓ Only owner should pause/unpause
✓ Owner should withdraw (247ms)
✓ Only owner should withdraw
✓ Should not withdraw to zero address (44ms)
✓ Owner should withdraw PLS (45ms)
✓ Only owner should withdraw
✓ Should not withdraw to zero address
✓ Should withdraw to valid account

Staking

Deployment

✓ Should set the right variables (6168ms)
✓ Should not initialize twice

Stake

✓ Should stake (2481ms)
✓ Should stake valid amount (247ms)

Withdraw

✓ Wthdraw when reward does not exist (408ms)

✓ Withdraw when reward exists (5187ms)
✓ Should claim after lockup (411ms)
✓ Should have valid grove
✓ Should withdraw valid value (234ms)
✓ Should withdraw after lockup (135ms)

Exit

✓ Exit when reward exists (361ms)

Allocate Seigniorage

✓ Should allocate (381ms)
✓ Only owner or treasury should allocate (633ms)
✓ Should allocate valid amount (230ms)
✓ Should have total supply when allocating

SetTreasuryOperator

✓ Set a new treasury operator (51ms)
✓ Only owner should set a new treasury operator

Set lockup

✓ Set a new lockup (53ms)
✓ Only owner should set a new lockup

Governance Recover

✓ Should recover (720ms)
✓ Only owner should recover
✓ Should not recover favor
✓ Should not recover esteem

One block test

✓ Should not stake twice in one block (2726ms)
✓ Should not withdraw twice in one block (93ms)
✓ Should not allocate twice in one block (1020ms)

Twap Oracle

Deployment

✓ Should set the right variables (4084ms)

Update

✓ Should update (526ms)
✓ Epoch should be updated (142ms)
✓ Epoch should be updated after multiple epochs (95ms)
✓ Set period (61ms)
✓ Set epoch

Consult

✓ Should consult when token0 (46ms)
✓ Should consult when token1 (52ms)
✓ Should revert when invalid token
✓ Should compare with max cap (89ms)

Twap

✓ Should twap when token0
✓ Should twap when token1
✓ Should return 0 when invalid token
✓ Should compare with max cap (65ms)

Operator

✓ isOperator
✓ TransferOperator
✓ Only owner TransferOperato
✓ Only TransferOperator to a valid address

FavorRouterWrapper

Deployment

✓ Should set the right variables (8832ms)

Add/Remove favor

✓ Should add favor (481ms)
✓ Should remove favor (48ms)

Set router

✓ Should set router (62ms)

Swap

✓ Should swapETHForFavorAndTrackBonus (1800ms)
✓ Should swapETHForFavorAndTrackBonus for only favor
✓ Should swapExactTokensForFavorAndTrackBonus (1187ms)
✓ Should swapExactTokensForFavorAndTrackBonus for only favor
✓ Should swapExactFavorForTokens (1299ms)
✓ Should swapExactFavorForTokens for only favor
✓ Should swapExactFavorForETH (124ms)
✓ Should swapExactFavorForETH for only favor

Suite result: ok. 181 passed; 0 failed; 0 skipped

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
Esteem.sol	100	100	100	100
PulseMinter.sol	100	94.05	100	100
favorPLS.sol	100	100	100	100
Staking.sol	100	90	100	100
FavorTreasury.sol	94.74	95.16	100	93.42
uniswapWraper.sol	100	95.45	100	100
UniTWAPOracle.sol	94.59	92.86	95.24	95.71
All files	**98.32**	**94.15**	**99.19**	**98.09**

We are grateful for the opportunity to work with the BetterBank team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the BetterBank team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo