



SMART CONTRACT AUDIT REPORT

for

BetterBank Protocol



Prepared By: Xiaomi Huang

PeckShield
October 2, 2025

Document Properties

Client	BetterBank
Title	Smart Contract Audit Report
Target	BetterBank
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 2, 2025	Xuxian Jiang	Final Release
1.0-rc	September 20, 2025	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About BetterBank	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Lack of ERC7702 Compatibility in Tax Exempt Check	12
3.2	Improved Favor Circulating Supply Calculation	14
3.3	Missing Leftover Refund During Liquidity Addition	15
3.4	Nonsafe LP-Redeemable USD Calculation in LPOracle	17
3.5	Accommodation of Non-ERC20-Compliant Tokens	18
3.6	Trust Issue of Admin Keys	20
4	Conclusion	23
	References	24

1 | Introduction

Given the opportunity to review the design document and related source code of the `BetterBank Protocol` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About BetterBank

`BetterBank` protocol is designed to create a self-sustaining reward and liquidity ecosystem. Users acquire `Esteem` by depositing assets like `PLS`, `PLSX`, or `DAI`, and can then stake `Esteem` to earn `Favor`, the primary tradable token. `Favor`'s supply expands algorithmically based on its time-weighted average price (`TWAP`), and sales of `Favor` are taxed, with proceeds funneled to the treasury and lending pools. When `Favor`'s price is low, buyers receive bonus `Favor` incentives. It has a built-in oracle system to ensure accurate pricing for minting, redemption, and reward calculations. Overall, the protocol creates a circular economy where external assets fuel `Esteem` minting, which drives `Favor` rewards and liquidity, while treasury mechanisms sustain long-term growth. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of BetterBank

Item	Description
Name	BetterBank
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 2, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- <https://github.com/grape-finance/BB-Custom-contracts.git> (43fba5e)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/grape-finance/BB-Custom-contracts.git> (8096359, abfc914)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.




comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `BetterBank` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	
Low	3	
Informational	1	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities, and 1 informational issue.

Table 2.1: Key BetterBank Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack of ERC7702 Compatibility in Tax Exempt Check	Business Logic	Confirmed
PVE-002	Low	Improved Favor Circulating Supply Calculation	Business Logic	Resolved
PVE-003	Medium	Missing Leftover Refund During Liquidity Addition	Business Logic	Resolved
PVE-004	Informational	Nonsafe LP-Redeemable USD Calculation in LPOracle	Time And State	Confirmed
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Lack of ERC7702 Compatibility in Tax Exempt Check

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Favor
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `Favor` tokens in `BetterBank` represent the base trading assets paired with different underlying tokens (`PLS`, `PLSX`, and `DAI`). These tokens implement a taxation and bonus system where the taxation logic is implemented through an overridden `_update()` function to detect contract transfers, apply the appropriate tax, and send the collected amount directly to the treasury address. In the process of analyzing the taxation logic, we notice the intended implementation of applying taxation for a contract destination should be improved.

In the following, we show the implementation of this specific function `_update()`. It has a rather straightforward logic in detecting whether the destination address is a contract (as well as the source/destination addresses are tax exempt) and applying the taxation accordingly. However, the way to detect whether the destination is a contract is based on `_to.code.length != 0`, which does not take into consideration of the ERC7702 specification. In particular, the ERC7702 specification allows any EOA to set its code based on any existing smart contract. To accommodate the ERC7702 specification, there is a need to improve the logic to reliably detect whether a given address is a true contract (excluding ERC7702-enabled EOAs).

```
102     function _update(address _from, address _to, uint256 _value) internal override {  
103         bool destinationIsContract = _to.code.length != 0;  
104  
105  
106         if (_isTaxExempt(_to) & _isTaxExempt(_from)) {  
107             super._update(_from, _to, _value);  
108         }  
109     }
```

```

108         return;
109     }
110
111     uint256 taxAmount = 0;
112
113     if (destinationIsContract) {
114         taxAmount = (_value * sellTax) / MULTIPLIER;
115     }
116
117     // tax goes to treasury
118     if (taxAmount > 0) {
119         super._update(_from, treasury, taxAmount);
120         _value -= taxAmount;
121     }
122
123     super._update(_from, _to, _value);
124 }

```

Listing 3.1: Favor::_update()

Recommendation Revise the above _update() function to properly detect whether a given address is a contract. An example implementation can be found as follows:

```

102     function _isContract(address who) private view returns (bool) {
103         uint256 csize = address(who).code.length;
104
105         // EOA
106         if (csize == 0) return false;
107
108         // Delegated EOA (EIP-7702)
109         if (csize == 23) {
110             bytes32 word;
111             assembly {
112                 let ptr := mload(0x40)
113                 extcodecopy(who, ptr, 0, 3) // copy first 3 bytes
114                 word := mload(ptr) // load the 32-byte word
115             }
116             return bytes3(word) != 0xef0100;
117         }
118
119         return true;
120     }

```

Listing 3.2: Favor::_isEOA()

Status This issue has been acknowledged.

3.2 Improved Favor Circulating Supply Calculation

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FavorTreasury
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The BetterBank protocol has a core FavorTreasury contract that implements an automated seigniorage system to manage token supply expansion based on the associated TWAP price. With that, there is a need to compute the circulating supply by excluding balances from excluded ones. While reviewing its implementation, we notice current logic can be improved.

In the following, we show the implementation of the related function, i.e., `getFavorCirculatingSupply()`. The current logic is rather straightforward: it basically deducts the balances of excluded addresses from current total supply. However, the exclusion of marked LP pair addresses makes the issue complicated. In particular, for each excluded LP pair, if it is already an excluded address, there is no need to double exclude the balance further (line 243). Moreover, current logic may be subject to possible sandwiching attack in having a skewed Favor balance in an excluded LP pair. However, the sandwiching attack may be mitigated with the inherent taxation logic when Favor are being transferred. With that, we need to be cautious when exempting certain LP pair from taxation.

```

226     function getFavorCirculatingSupply() public view returns (uint256) {
227         uint256 totalSupply = IERC20(favor).totalSupply();
228
229         // Loose Favor balances of excluded addresses
230         uint256 balanceExcluded = 0;
231         uint256 excludedLen = excludedAddresses.length;
232         for (uint256 i = 0; i < excludedLen; i++) {
233             balanceExcluded += IERC20(favor).balanceOf(excludedAddresses[i]);
234         }
235
236         // Favor in LP tokens held by excluded addresses
237         uint256 favorFromLpHeldByExcluded = 0;
238         uint256 pairsLen = lpPairsToExclude.length;
239
240         // Loop through LP pairs excluded and calculate Favor reserves in each
241         for (uint256 j = 0; j < pairsLen; j++) {
242             address pair = lpPairsToExclude[j];
243             if (!isLpPairToExclude[pair]) continue;
244
245             IUniswapV2Pair p = IUniswapV2Pair(pair);
246
247             (uint112 r0, uint112 r1, ) = p.getReserves();

```

```

248         address t0 = p.token0();
249         uint256 favorReserves = (t0 == favor) ? uint256(r0) : uint256(r1);
250
251         uint256 lpTotal = p.totalSupply();
252         if (lpTotal == 0 || favorReserves == 0) continue;
253
254         // Get excluded LP token balance of each excluded address and add to
           excluded balance of Favor
255         for (uint256 i = 0; i < excludedLen; i++) {
256             uint256 holderLp = p.balanceOf(excludedAddresses[i]);
257             if (holderLp == 0) continue;
258             favorFromLpHeldByExcluded += (favorReserves * holderLp) / lpTotal;
259         }
260     }
261
262     return totalSupply - balanceExcluded - favorFromLpHeldByExcluded;
263 }

```

Listing 3.3: FavorTreasury::getFavorCirculatingSupply()

Recommendation Revise the above `getFavorCirculatingSupply()` function to reliably and accurately compute the circulating supply.

Status This issue has been fixed in the following commit: `abfc914`.

3.3 Missing Leftover Refund During Liquidity Addition

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: LPZapper
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

To facilitate user interaction, the BetterBank protocol has provided a Zapper contract, which provides gated trading and liquidity management functionality, integrating with both `PulseX` router functions and BetterBank lending pools. In this Zapper contract, we notice the presence of two liquidity-adding functions, i.e., `addLiquidity()` and `addLiquidityETH()`. Our analysis shows these two functions can be improved by always refunding leftover funds, if any, after adding the intended liquidity.

To elaborate, we use the `addLiquidity()` function as an example and show below its implementation. We notice the tokens to add are transferred to the Zapper contract before interacting with the router to add liquidity. When the provided tokens are not balanced, it is likely one token may be used up and another is not. In this case, the unused funds after the liquidity addition should

be properly refunded back to the user, which is missing in current implementation. Note another `addLiquidityETH()` routine shares the same issue.

```

283     function addLiquidity(
284         address tokenA,
285         address tokenB,
286         uint amountADesired,
287         uint amountBDesired,
288         uint amountAMin,
289         uint amountBMin,
290         address to,
291         uint deadline
292     ) external {
293         require(favorToToken[tokenA] == tokenB, "Zapper: Not listed to make LP");
294         IERC20(tokenA).transferFrom(msg.sender, address(this), amountADesired);
295         IERC20(tokenB).transferFrom(msg.sender, address(this), amountBDesired);
296
297         IERC20(tokenA).forceApprove(address(router), amountADesired);
298         IERC20(tokenB).forceApprove(address(router), amountBDesired);
299
300         router.addLiquidity(
301             tokenA,
302             tokenB,
303             amountADesired,
304             amountBDesired,
305             amountAMin,
306             amountBMin,
307             to,
308             deadline
309         );
310     }

```

Listing 3.4: LPZapper::addLiquidity()

Recommendation Revise the above-mentioned routines to properly refund leftover funds back to the user.

Status This issue has been fixed in the following commit: 4f19f74.

3.4 Nonsafe LP-Redeemable USD Calculation in LPOracle

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LPOracle
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

The BetterBank protocol supports the use of liquidity pool tokens and has a specialized LP oracle contract, i.e., `LPTWAPOracle`, to calculate USD-denominated values. In the process of reviewing the getter functions to compute raw redeemable USD per LP in the format of `Q112.112` or `1e18` scale, we notice current computation may be affected from possible MEV-based sandwiching attacks.

In particular, we show below the implementation of the related routines, i.e., `redeemableUsdPerLpQ112()` and `redeemableUsdPerLpScaled()`. Their logic is straightforward by querying spot reserves, then making use of the average prices of respective tokens, and finally computing the raw redeemable USD per LP. While the prices are relatively reliable in not using the spot prices, the spot reserves may be problematic and could be affected with large swaps, resulting in imbalanced pools. With that, we need to exercise cautious when using these two routines.

```

235     function redeemableUsdPerLpQ112() public view returns (uint256) {
236         (uint112 r0, uint112 r1,) = pair.getReserves();
237         uint256 px0 = uint256(usd0Average._x);
238         uint256 px1 = uint256(usd1Average._x);
239         uint256 totalUsdQ = uint256(r0) * px0 + uint256(r1) * px1;
240         return totalUsdQ / pair.totalSupply();
241     }
242
243     /// @notice Redeemable USD per LP scaled by 1e18
244     function redeemableUsdPerLpScaled() public view returns (uint144) {
245         uint256 q112 = redeemableUsdPerLpQ112();
246         uint256 scaled = q112 * 1e18 >> 112;
247         return uint144(scaled);
248     }

```

Listing 3.5: `LPOracle::redeemableUsdPerLpQ112()/redeemableUsdPerLpScaled()`

Recommendation Revise the above-mentioned routines to reliably compute the raw redeemable USD per LP in desired formats.

Status This issue has been acknowledged.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ftYieldWrapper, PutManager
- Category: Coding Practice [7]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (<https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>).

```

194  /**
195   * @dev Approve the passed address to spend the specified amount of tokens on behalf
       of msg.sender.
196   * @param _spender The address which will spend the funds.
197   * @param _value The amount of tokens to be spent.
198   */
199   function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201       // To change the approve amount you first have to reduce the addresses '
202       // allowance to zero by calling 'approve(_spender, 0)' if it is not
203       // already 0 to mitigate the race condition described here:
204       // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205       require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

207       allowed[msg.sender][_spender] = _value;
208       Approval(msg.sender, _spender, _value);
209   }

```

Listing 3.6: USDT Token Contract

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```

38  /**
39   * @dev Deprecated. This function has issues similar to the ones found in
40   * {IERC20-approve}, and its usage is discouraged.
41   *
42   * Whenever possible, use {safeIncreaseAllowance} and
43   * {safeDecreaseAllowance} instead.
44   */
45  function safeApprove(
46      IERC20 token,
47      address spender,
48      uint256 value
49  ) internal {
50      // safeApprove should only be called when setting an initial allowance,
51      // or when resetting it to zero. To increase and decrease it, use
52      // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
53      require(
54          (value == 0) (token.allowance(address(this), spender) == 0),
55          "SafeERC20: approve from non-zero to non-zero allowance"
56      );
57      _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
58          spender, value));
59  }

```

Listing 3.7: SafeERC20::safeApprove()

In current implementation, if we examine the Zapper::_swap() routine that is designed to facilitate token swaps. To accommodate the specific idiosyncrasy, there is a need to use safeApprove(), instead of approve() (line 174). Note another routine addLiquidity() in the same contract shares the same issue.

```

168  function _swap(
169      address _in,
170      address _out,
171      uint256 _amount,
172      uint256 _deadline
173  ) internal returns (uint256) {
174      IERC20(_in).approve(address(router), _amount);
175
176      address[] memory path = new address[](2);
177      path[0] = _in;
178      path[1] = _out;
179
180      uint256 before = IERC20(_out).balanceOf(address(this));
181      router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
182          _amount,
183          0,
184          path,
185          address(this),
186          _deadline
187      );
188      uint256 got = IERC20(_out).balanceOf(address(this)) - before;
189  }

```

```

190     require(got > 0, "Zapper: Swap failed");
191     return got;
192 }

```

Listing 3.8: Zapper::_swap()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve().

Status This issue has been fixed in the following commit: 9adebe6.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the BetterBank protocol, there is a privileged account `owner` that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters, manage minters, recover funds, and execute privileged operations). The account also has the privilege to control or govern the flow of assets managed by this contract. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```

204     function setApprovedUser(address user, bool allowed) external onlyOwner {
205         require(user != address(0), "Zero address not allowed");
206         isApprovedUser[user] = allowed;
207         emit ApprovedUserSet(user, allowed);
208     }
209
210     function setDailyRateIncrease(uint256 _newRate) external onlyOwner {
211         dailyRateIncrease = _newRate;
212         emit NewDailyRateIncrease(_newRate);
213     }
214
215     function setEsteemRate(uint256 _rate) external onlyOwner {
216         require(_rate > 0, "Esteem Rate must be > 0");
217         esteemRate = _rate;
218         emit RateUpdated(_rate);
219     }
220
221     function setPool(address _pool) external onlyOwner {

```

```

222     require(_pool != address(0), "Must be a valid address");
223     POOL = IPool(_pool);
224     emit NewPOOL(_pool);
225 }
226
227 function setRedeemRate(uint256 _redeemRate) external onlyOwner {
228     require(_redeemRate <= MULTIPLIER, "Cannot exceed 100%");
229     redeemRate = _redeemRate;
230     emit RedeemRateUpdated(_redeemRate);
231 }
232
233 function setTreasuryBonus(uint256 _treasuryBonusRate) external onlyOwner {
234     require(_treasuryBonusRate <= MULTIPLIER, "Cannot exceed 100%");
235     treasuryBonusRate = _treasuryBonusRate;
236     emit TreasuryBonusUpdated(_treasuryBonusRate);
237 }
238
239 function setTreasury(address _holding, address _team) external onlyOwner {
240     require(_holding != address(0), "Invalid Holding address");
241     require(_team != address(0), "Invalid Team address");
242     holding = _holding;
243     team = _team;
244     emit TreasuryUpdated(_holding, _team);
245 }
246
247 function setPriceOracle(address token, address oracle) external onlyOwner {
248     require(oracle != address(0), "Invalid oracle");
249     priceOracles[token] = oracle;
250     emit OracleUpdated(token, oracle);
251 }
252
253 function setAllowedMintToken(address token, bool allowed) external onlyOwner {
254     require(token != address(0), "Invalid token");
255     allowedMintTokens[token] = allowed;
256     emit AllowedMintTokenSet(token, allowed);
257 }
258
259 function setActiveFavorToken(address token, bool allowed) external onlyOwner {
260     require(token != address(0), "Invalid token");
261     favorTokens[token] = allowed;
262     emit ActiveFavorTokenSet(token, allowed);
263 }
264
265 // Admin withdraw incase of stuck or mistakenly sent tokens, no user tokens are
    stored in this contract
266 function adminWithdraw(IERC20 _token, address _to, uint256 _amount) external
    onlyOwner {
267     require(_to != address(0), "Invalid address");
268     _token.safeTransfer(_to, _amount);
269     emit AdminWithdraw(address(_token), _to, _amount);
270 }
271

```

```
272     function adminWithdrawPLS(address _to, uint256 _amount) external onlyOwner {
273         require(_to != address(0), "Invalid address");
274         (bool success, ) = _to.call{value: _amount}("");
275         require(success, "Transfer failed");
276         emit AdminWithdraw(address(0), _to, _amount);
277     }
278
279     function pause() external onlyOwner {
280         _pause();
281         emit ContractPaused(msg.sender);
282     }
283
284     function unpause() external onlyOwner {
285         _unpause();
286         emit ContractUnpaused(msg.sender);
287     }
```

Listing 3.9: Example Privileged Functions in `MintRedeemer`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the BetterBank protocol, which is designed to create a self-sustaining reward and liquidity ecosystem. Users acquire Esteem by depositing assets like PLS, PLSX, or DAI, and can then stake Esteem to earn Favor, the primary tradable token. Favor's supply expands algorithmically based on its time-weighted average price (TWAP), and sales of Favor are taxed, with proceeds funneled to the treasury and lending pools. When Favor's price is low, buyers receive bonus Favor incentives. It has a built-in oracle system to ensure accurate pricing for minting, redemption, and reward calculations. Overall, the protocol creates a circular economy where external assets fuel Esteem minting, which drives Favor rewards and liquidity, while treasury mechanisms sustain long-term growth. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [5] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [6] MITRE. CWE CATEGORY: 7PK - Time and State. <https://cwe.mitre.org/data/definitions/361.html>.
- [7] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [8] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [9] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.

- [10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [11] PeckShield. PeckShield Inc. <https://www.peckshield.com>.

