SMART CONTRACT AUDIT REPORT

for

Monroe Protocol

Prepared By: Xiaomi Huang

PeckShield

March 4, 2024

Document Properties

Client	GoodEntry
Title	Smart Contract Audit Report
Target	Monroe Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 4, 2024	Xuxian Jiang	Final Release
1.0-rc	February 29, 2024	Xuxian Jiang	Release Candidate

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 Monroe . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 7

2 Findings 9

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results 11

3.1 Revisited HealthRate Calculation in BaseVault . . . 11

3.2 Timely And Accurate Income Collection in BaseVault . . . 12

3.3 Incorrect Deposit Accounting in EmergencyPool . . . 14

3.4 Accommodation of Non-ERC20-Compliant Tokens . . . 15

3.5 Trust Issue of Admin Keys . . . 17

4 Conclusion 19

References 20

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Monroe 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 Monroe

Monroe is a new DeFi primitive built on realising the full potential of liquid staking tokens (LSTs) across all EVM compatible chains. It achieves this by enabling the creation of stablecoins from LSTs in a fully decentralized way. The protocol makes incremental innovations on the back of giants such as Liquity, Lybra and Prisma. The envisioned outcome is that these stablecoins will be able to maintain its peg without significant price variance in different market conditions. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Monroe Protocol

Item	Description
Target	Monroe Protocol
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	March 4, 2024

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Note that the Monroe protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/MonroeProtocol/contracts.git (fe5b79a)

And these are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/MonroeProtocol/contracts.git (bf902c4)

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

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 classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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.

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 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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Monroe implementation. 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 logic, 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	3	■ ■ ■
Low	2	■ ■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. 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 that 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, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Monroe Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Health Factor Calculation in BaseVault	Business Logic	Fixed
PVE-002	Medium	Timely And Accurate Income Collection in BaseVault	Time and State	Fixed
PVE-003	Low	Incorrect Deposit Accounting in EmergencyPool	Business Logic	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Revisited HealthRate Calculation in BaseVault

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: BaseVault
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

In the Monroe protocol, there is a core BaseVault contract that underpins the implementation of various vaults by recording user collateral and debt. While examining the associated health factor calculation from user collateral and debt, we notice current approach needs to take into account the decimals of underlying collateral and debt.

To elaborate, we show below the related getHealthFactor() routine. It has a rather straightforward logic in computing a user's health factor based on the following formula, i.e., collateraValue * 100 / debtValue (line 242). However, the collateraValue calculation is computed as balanceOf(user)* latestPrice(), which needs to be revised as balanceOf(user)* latestPrice()/ 2**IERC20Upgradeable(collateralAsset()).decimals(). Similarly, the debt vaule needs to be revised as _debtBalance * ISynth (synth).getPrice()/ 2**ISynth(synth).decimals(). And the final health factor can then be computed as hf = 100 * balanceOf(user)* latestPrice()* 2**ISynth(synth).decimals()/ _debtBalance / ISynth (synth).getPrice()/ 2**IERC20Upgradeable(collateralAsset()).decimals().

```
240  function getHealthFactor(address user) public view returns (uint hf){
241    uint _debtBalance = debtOf(user);
242    if (_debtBalance > 0) hf = 100 * balanceOf(user) * latestPrice() / _debtBalance /
         ISynth(synth).getPrice();
243    else hf = type(uint).max;
244  }
```

Listing 3.1: BaseVault::getHealthFactor()

Recommendation Revisit the above routine to properly the user's health factor. Note the same issue also affects other routines, including `_liquidateCollateral()`, `emergencyRepay()`, and `rigidRedemption()`.

Status The issue has been addressed in the following commits: `0d62223`, `599c659`, and `bf902c4`.

3.2 Timely And Accurate Income Collection in BaseVault

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BaseVault`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As mentioned in Section 3.1, the `Monroe` protocol has a core `BaseVault` contract that underpins the implementation of various vaults. Each vault may have its income that needs to be properly collected for overall collateral and index adjustment. While analyzing the income collection, we notice the income needs to timely and accurately collected and distributed.

In the folowing, we examine the `BaseVault` contract and report an issue in `emergencyRepay()` that does not timely collect the income. Specifically, this `emergencyRepay()` routine allows to repay the debt in the emergency pool and get respective collateral in return. However, the logic needs to invoke `checkIncome()` before making any debt payment.

```
function emergencyRepay(uint debtAmount) external {
  uint repaidValue = debtAmount * ISynth(synth).getPrice();
  uint clawedAmount = repaidValue * 108 / 100 / latestPrice();
  ISynth(synth).burn(msg.sender, debtAmount);
  decreaseDebt(emergencyPool, debtAmount);
  IERC20Upgradeable(collateralAsset()).safeTransfer(msg.sender, clawedAmount);
  decreaseCollateral(emergencyPool, clawedAmount);
  _totalDepositedCollateral -= clawedAmount;

  emit RepayEmergencyDebt(debtAmount, clawedAmount);
}
```

Listing 3.2: `BaseVault::emergencyRepay()`

In addition, the derived `RebaseCollateralVault` contract from `BaseVault` has a concrete `checkIncome()` implementation. Our analysis shows its implementation can be improved. Specifically, the internal state `epShare` records the emergency pool share of new income that will be credited to `emergencyPool` as a new deposit into the vault. With that, there is a need to update `_totalDepositedCollateral` as follows: `_totalDepositedCollateral += epShare` (line 24). Similarly, another derived `RebaseCollateralVault` contract shares the same issue.

```
18 function checkIncome() public override returns (uint income){
19   uint actualBal = ERC20(collateralAsset()).balanceOf(address(this));
20   if (actualBal > _totalDepositedCollateral){
21     income = actualBal - _totalDepositedCollateral;
22     if (balanceOf(emergencyPool) > 0){
23       uint emergencyPoolShareTarget = controller.emergencyPoolShare();
24       uint epShare = income * emergencyPoolShareTarget * 15 / 1000;
25       increaseCollateral(emergencyPool, epShare);
26       income -= epShare;
27     }
28     // part of the income distributed out (to treasury and savings)
29     income -= _distributeIncome(income);
30     /*
31       Remaining income is given to distributors thru rebase (increase liquidityIndex)
32       totalBalances * liquidityIndex = _totalDepositedCollateral
33       new_totalDepositedCollateral = old_totalDepositedCollateral + income
34 
35       since balances dont change:
36       newLiquidityIndex / new_totalDepositedCollateral = old_liquidityIndex / old_totalDepositedCollateral
37     */
38     liquidityIndex = (_totalDepositedCollateral + income) * liquidityIndex / _totalDepositedCollateral;
39     // sanity check (cant require or error would brick the vault): _totalDepositedCollateral + income = ERC20(collateralAsset()).balanceOf(address(this))
40     _totalDepositedCollateral = ERC20(collateralAsset()).balanceOf(address(this));
41     emit CollectIncome(income);
42   }
43 }
```

Listing 3.3: `RebaseCollateralVault::checkIncome()`

Recommendation Revise the above-mentioned routines to timely and properly update new income.

Status The issue has been addressed in the following commits: `94bf1f9` and `bf902c4`.

3.3 Incorrect Deposit Accounting in EmergencyPool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `EmergencyPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

The `Monroe` protocol has the notion of `EmergencyPool` that keeps debt from emergency liquidation. This `EmergencyPool` contract is implemented as an `ERC4626` vault with a standard API for tokenized yield-bearing vaults, offering basic functionality for depositing, withdrawing tokens, and reading balances. In the process of examining the related deposit logic, we notice the implementation makes an extension and that extension can be improved.

To elaborate, we show below the related code snippet of the `_deposit()` routine. The purpose here is to deposits assets of underlying tokens into the vault and grants ownership of shares to receiver. The `EmergencyPool` contract extends the logic by also keeping track of the `depositTime` of caller. In fact, the `depositTime` state should be about the `receiver`, not `msg.sender` (line 43).

```
40  /// @notice Forward assets to collateral vault after deposit
41  function _deposit(address caller, address receiver, uint256 assets, uint256 shares)
        internal override {
42    super._deposit(caller, receiver, assets, shares);
43    depositTime[msg.sender] = block.timestamp;
44    IERC20(asset()).approve(collateralVault, assets);
45    IBaseVault(collateralVault).depositAndMint(assets, 0);
46  }
```

Listing 3.4: `EmergencyPool::_deposit()`

Moreover, the `depositTime` state is used to detect early withdrawals. An early withdrawal situation may charge 0.1% fee. However, this detection can be easily bypassed as the share can be transferred to a new fresh account to perform the actual withdrawal.

```
/// @notice Withdraw assets from collateral vault before transfer and apply 0.1%
    sniping penalty
function _withdraw(address caller, address receiver, address owner, uint256 assets,
    uint256 shares) internal override {
  IBaseVault(collateralVault).withdrawAndBurn(assets, 0);
  if (block.timestamp < depositTime[owner] + 1 days) assets = assets * 999 / 1000;
  super._withdraw(caller, receiver, owner, assets, shares);
}
```

Listing 3.5: `EmergencyPool::_withdraw()`

Recommendation Revise the above routine by properly keeping track of the `depositTime` state and reliably charge the early withdrawal fee.

Status The issue has been addressed in the following commits: `d03fdf4` and `b020f1c`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [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).

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

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

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

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()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.7: `SafeERC20::safeApprove()`

In current implementation, if we examine the `BaseVault::_distributeIncome()` routine that is designed to distribute new income. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (line 300). And it is better to be invoked twice: the first `safeApprove()` resets the spending allowance and the second sets up the intended allowance.

```
  function _distributeIncome(uint amount) internal returns (uint distributed) {
    uint treasuryFee        = controller.treasuryFee();
    uint userShare          = 10_000 - treasuryFee;
    // After fees, split the income between depositors and savings
    uint shareSavings  = userShare * ISynth(synth).getSavingsYield() / 10_000;
    uint treasuryAmount = amount * treasuryFee / 10_000;
    IERC20Upgradeable(collateralAsset()).safeTransfer(controller.treasury(),
        treasuryAmount);
    amount -= treasuryAmount;

    // Send its share to savings pool for Dutch auction
    uint savingsAmount = amount * shareSavings / 10_000;
    if(IERC20Upgradeable(collateralAsset()).allowance(address(this), synth) <
        savingsAmount) IERC20Upgradeable(collateralAsset()).approve(synth, savingsAmount
        );
```

```
  ISynth(synth).collectSavingsIncome(collateralAsset(), savingsAmount);

  distributed = treasuryAmount + savingsAmount;
}
```

Listing 3.8: `BaseVault::_distributeIncome()`

Note the `EmergencyPool::_deposit()` routine can be similarly improved.

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

Status The issue has been addressed in the following commit: `a1df7e3`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Monroe` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting, vault adjustment, and synth creation). 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.

```
function setTreasury(address _treasury) public onlyOwner {
  require(_treasury != address(0), "Ctrl: Null Address");
  treasury = _treasury;
}


/// @notice Add a new collateral vault
function addVault(address vault) public onlyOwner returns (uint) {
  require(vault != address(0), "Ctrl: Invalid Vault");
  address collateral = IBaseVault(vault).collateralAsset();
  address oracle = IBaseVault(vault).oracle();
  require(collateral != address(0) && oracle != address(0), "Ctrl: Invalid Vault");
  require(collateralToVault[collateral] == address(0), "Ctrl: Vault Already Exists");

  vaults.push(vault);
  collateralToVault[collateral] = vault;
  return vaults.length;
}
```

```
...
function createSynth(bytes32 name, address oracle) public onlyOwner returns (address
    newSynth) {
  if (oracle != address(0)) require(AggregatorInterface(oracle).latestAnswer() > 0, "
      Ctrl: No Such Oracle");
  string memory _name = string(abi.encodePacked("Monroe", name));
  string memory _symbol = string(abi.encodePacked(name, "m"));
  newSynth = Clones.clone(synth);
  ISynth(newSynth).initialize(_name, _symbol, oracle, lz0Endpoint, savingsPoolLogic);
  synths.push(newSynth);
  ISynth(newSynth).transferOwnership(msg.sender);
}
```

Listing 3.9: Privileged Operations in `Controller`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed by the team. For the time being, it is planned to mitigate with a timelock mechanism.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Monroe protocol, which is a new DeFi primitive built on realising the full potential of liquid staking tokens (LSTs) across all EVM compatible chains. It achieves this by enabling the creation of stablecoins from LSTs in a fully decentralized way. The protocol makes incremental innovations on the back of giants such as Liquity , Lybra and Prisma. The envisioned outcome is that these stablecoins will be able to maintain its peg without significant price variance in different market conditions. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.