BEOSIN
Blockchain Security

Monroe Protocol

Smart Contract Security Audit

No. 202403131737

Mar 13th, 2024

SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Result

After auditing, 2 Medium-risk and 3 Low-risk items were identified in the Monroe Protocol. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:

Business overview

1. Business overview

Monroe Protocol is a collateralized lending project primarily comprised of the following components: EmergencyPool, SavingsPool, Synth, Vault, and Controller. Each module will be described in detail below.

EmergencyPool: The contract inherits from ERC4626 contract to specify tokens as collateral assets, and users can deposit them using the `deposit` function. EmergencyPool then pledges these assets to the Vault for collateralization. Users who deposit assets into EmergencyPool can benefit from the income generated by the Vault during liquidation and at the end of the income period. It's important to note the following:

1. Users participating in EmergencyPool deposits will incur a fee of one-thousandth each time if they perform withdrawal or transfer operations within the first day of depositing.

2. Additionally, users should be aware that because the overall assets of EmergencyPool increase the total debt during liquidation in the Vault, there may be situations where users are unable to withdraw funds due to debt checks if they withdraw large amounts.

SavingsPool: This contract inherits functionality from the ERC4626Upgradeable contract and uses Synth contract tokens as assets. Users can participate in deposits using Synth tokens in this contract. After settling Income, the Vault transfers a portion of the Income (with collateralAsset as the asset type) to the SavingsPool to initiate an Auction. Other users can bid on these collateralAssets using Synth contract tokens. The Synth tokens spent on purchases are included in the contract's fees. Users who participated in deposits in the SavingsPool contract earlier can enjoy dividends accumulated from fees one day prior. After the Auction begins, the auction price of collateralAssets will discount over time, reaching a discount cap of 50% after 30 minutes.

Synth: The contract is deployed with a binding to the SavingsPool and can set two liquidation ratios, hardLiquidationThreshold and liquidationThreshold. It can also synchronize the deployment of the Vault contract with the Controller to maintain consistency. When there is a change in the debt of the Vault contract, the Synth contract will mint or burn the corresponding token amount.

Vault: Each Vault contract is deployed with bindings to the EmergencyPool and Synth contracts. The Vault can provide the `depositAndMint` function to users and the EmergencyPool for collateralization. Users can collateralize the specified collateralAsset into the Vault to increase collateralBalances for borrowing Synth tokens. When withdrawing, if a user has debt, the health factor must be greater than

160 after withdrawal. In the event of a decrease in collateralAsset price, there are two ways to liquidate. The first involves users using their own synth tokens to eliminate the debt of the liquidated party, earning a profit of 1% to 9% in collateralAsset. The second uses EmergencyPool assets for liquidation, where the user earns a profit of 1% in collateralAsset, with the remaining 8% going to the EmergencyPool. Both liquidation methods incur a 1% fee sent to the treasury address. Users can also use the `rigidRedemption` function to repay debt with synth tokens and withdraw collateralAsset (the contract charges a 0.5% fee).

Controller: The owner of the Controller contract has the authority to add Vault contracts to expand the lending market (which will later be synchronized through the Synth contract). The owner also has the permission to modify the values of liquidationThreshold and hardLiquidationThreshold in the Synth contract (used for Vault's liquidation checks).

1 Overview

1.1 Project Overview

Project Name	Monroe Protocol
Project Language	Solidity
Platform	Ethereum、Manta Pacific、Avax
Audit Scope	https://github.com/Monroe Protocol/contracts/commits/main/contracts
Commit Hash	6b5b54088c3621145cf7e5a252f6efef773144ef(Initial) 38f3386abe19c0ccc45886e48dd9a5b8b2fb1e68 bfb1ae4caa5ea08b0e04b88a9e34584e07730a0f(Final)

1.2 Audit Overview

Audit work duration: Mar 8, 2024 – Mar 13, 2024

Audit team: Beosin Security Team

1.3 Audit Method

The audit methods are as follows:

1. Formal Verification

Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using Beosin's library of security expert rules. These rules call into the contracts under analysis and make various assertions about their behavior. The rules of the specification play a crucial role in the analysis. If the rule is violated, a concrete test case is provided to demonstrate the violation.

2. Manual Review

Using manual auditing methods, the code is read line by line to identify potential security issues. This ensures that the contract's execution logic aligns with the client's specifications and intentions, thereby safeguarding the accuracy of the contract's business logic.

The manual audit is divided into three groups to cover the entire auditing process:

The Basic Testing Group is primarily responsible for interpreting the project's code and conducting comprehensive functional testing.

The Simulated Attack Group is responsible for analyzing the audited project based on the collected historical audit vulnerability database and security incident attack models. They identify potential attack vectors and collaborate with the Basic Testing Group to conduct simulated attack tests.

The Expert Analysis Group is responsible for analyzing the overall project design, interactions with third parties, and security risks in the on-chain operational environment. They also conduct a review of the entire audit findings.

3. Static Analysis

Static analysis is a method of examining code during compilation or static analysis to detect issues. Beosin-VaaS can detect more than 100 common smart contract vulnerabilities through static analysis, such as reentrancy and block parameter dependency. It allows early and efficient discovery of problems to improve code quality and security.

2 Findings

Index	Risk description	Severity level	Status
Monroe Protocol-01	The getPrice function will always return 1e8	Medium	Fixed
Monroe Protocol-02	Missing health factor check after liquidation	Medium	Partially Fixed
Monroe Protocol-03	The function lacks validation for oracleToken1	Low	Fixed
Monroe Protocol-04	Funds locked in contract Inaccessible for withdrawal	Low	Fixed
Monroe Protocol-05	DOS vulnerability in depositTime setting	Low	Acknowledged

Finding Details:

[Monroe Protocol-01] The getPrice function will always return 1e8

Severity Level	Medium
Type	Business Security
Lines	Synth.sol #L113-118
Description	In the Synth contract, the `getPrice` function always returns priceX8 as 1e8. Although AggregatorInterface(oracle).`latestAnswer`() is called, the returned price is not updated to the priceX8 variable, resulting in `getPrice` always returning 1e8 unchanged.
Recommendation	It is recommended to use `priceX8` to store the return value of AggregatorInterface(oracle).`latestAnswer`().
Status	**Fixed**.

```
  /// @notice Get target price of the synth, 1e8 in case of USD synth
  /// @dev Override with a call to the oracle for a non USD synth
  function getPrice() public virtual view returns (uint priceX8){
    priceX8 = 1e8;
    if (oracle != address(0))
AggregatorInterface(oracle).latestAnswer();
  }
```

```
  /// @notice Get target price of the synth, 1e8 in case of USD synth
  /// @dev Override with a call to the oracle for a non USD synth
  function getPrice() public virtual view returns (uint priceX8){
    priceX8 = 1e8;
    if (oracle != address(0)) priceX8 =
uint(AggregatorInterface(oracle).latestAnswer());
  }
```

[Monroe Protocol-02] Missing health factor check after liquidation

Severity Level	Medium
Type	Business Security
Lines	BaseVault.sol #L222-246
Description	After the `liquidate` and `emergencyLiquidate` functions are called, there is no check to verify if the user or emergencyPool has defaulted. In extreme cases, this could lead to a situation where the liquidator uses a sufficient amount of debtAmount to reduce the user's collateralBalances to a minimal value after liquidation, yet the user still has outstanding debt. This would result in the project losing money to pay the liquidator's reward, while the defaults of the user and emergencyPool persist.

```
  function liquidate(address user, uint debtAmount) external {
    uint liquidatedAmount = _liquidateCollateral(user, debtAmount,
false);

    // Burn debt and synth
    ISynth(synth).burn(msg.sender,  debtAmount);
    decreaseDebt(user, debtAmount);
    emit Liquidate(user, debtAmount,  liquidatedAmount);
  }


  /// @notice Emergency liquidation only transfers debt to the emergency
pool
  /// @dev In case no synth available for liquidations or gas too high
  /// @dev Liquidator only gets a small fee to pay for gas
  function emergencyLiquidate(address user, uint debtAmount) external
{
    uint liquidatedAmount = _liquidateCollateral(user, debtAmount,
true);

    // Move debt to emergency pool
    decreaseDebt(user, debtAmount);
    increaseDebt(emergencyPool, debtAmount);
    emit EmergencyLiquidate(user, debtAmount, liquidatedAmount);
  }
```

Recommendation	It is recommended to add a check for the user's health factor after the `liquidate` and `emergencyLiquidate` functions. Additionally, it is suggested to include a check for the emergency pool's ledger after the `emergencyLiquidate` function.
Status	**Partially Fixed**. According to the project team's description, don't add a check at the end of a user liquidation because if a position is too large, it would become impossible to liquidate it in parts.

```
  /// @notice Emergency liquidation only transfers debt to the emergency
pool
  /// @dev In case no synth available for liquidations or gas too high
  /// @dev Liquidator only gets a small fee to pay for gas
  function emergencyLiquidate(address user, uint debtAmount) external
{
    uint liquidatedAmount = _liquidateCollateral(user, debtAmount,
true);

    // Move debt to emergency pool
    decreaseDebt(user, debtAmount);
    increaseDebt(emergencyPool, debtAmount);
    // Check health: the emergency pool HF should be more conservative
when hard liquidations happen
    require(getHealthFactor(emergencyPool) >=
ISynth(synth).liquidationThreshold() + 100, "BV: Unhealthy position");
    emit EmergencyLiquidate(user, debtAmount, liquidatedAmount);
  }
```

[Monroe Protocol-03] The function lacks validation for oracleToken1

Severity Level	Low
Type	Business Security
Lines	OracleConvert.sol #L262-278
Description	The OracleConvert contract contains two checks for the zero address of `_oracleToken0` but lacks a similar check for `_oracleToken1`. This omission results in the `_oracleToken1` variable being written without validation, which impacts the return value of latestAnswer. `constructor(address _oracleToken0, address _oracleToken1) { require(_oracleToken0 != address(0x0) && _oracleToken0 != address(0x0), "Invalid address"); require(AggregatorInterface(_oracleToken0).decimals() == 18 && AggregatorInterface(_oracleToken1).decimals() == 8, "Invalid decimals"); oracleToken0 = AggregatorInterface(_oracleToken0); oracleToken1 = AggregatorInterface(_oracleToken1); }`
Recommendation	It is recommended to replace one of the zero address checks for `_oracleToken0` with a check for `_oracleToken1`.
Status	**Fixed**. `constructor(address _oracleToken0, address _oracleToken1) { require(_oracleToken0 != address(0x0) && _oracleToken1 != address(0x0), "Invalid address"); require(AggregatorInterface(_oracleToken0).decimals() == 18 && AggregatorInterface(_oracleToken1).decimals() == 8, "Invalid decimals"); oracleToken0 = AggregatorInterface(_oracleToken0); oracleToken1 = AggregatorInterface(_oracleToken1); }`

[Monroe Protocol-04] Funds locked in contract Inaccessible for withdrawal

Severity Level	Low
Type	Business Security
Lines	EmergencyPool.sol #L34-39

Description	The `_withdraw` function in the EmergencyPool contract leaves 0.1% of the assets (`_collateralAsset`) in the EmergencyPool contract upon invocation, making them inaccessible for 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);`
	`}`

Recommendation	It is recommended to add a fee withdrawal function.

Status	**Fixed**.
	`/// @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 {`
	`if (block.timestamp < depositTime[owner] + 1 days) assets = assets * 999 / 1000;`
	`IBaseVault(collateralVault).withdrawAndBurn(assets, 0);`
	`super._withdraw(caller, receiver, owner, assets, shares);`
	`}`

[Monroe Protocol-05] DOS vulnerability in depositTime setting

Severity Level	Low
Type	Business Security
Lines	EmergencyPool.sol #L42-49
Description	In the `_deposit` function of the EmergencyPool contract, the `depositTime` parameter is set to receiver, allowing for deposits of negligible amounts to any receiver address. Consequently, when transactions such as transfers and withdrawals are made from the receiver address, they will be subject to a 0.1% fee.
Recommendation	It is recommended to add a minimum value for collateral in the `_deposit` function to prevent malicious manipulation of user deposit times.
Status	**Acknowledged**.

```
  /// @notice Forward assets to collateral vault after deposit
  function _deposit(address caller, address receiver, uint256 assets,
uint256 shares) internal override {
    // prevent depositTime griefing by only authorizing deposit for
oneself
    require(caller == receiver, "SP: Caller is not receiver");
    super._deposit(caller, receiver, assets, shares);
    depositTime[receiver] = block.timestamp;
    IERC20(asset()).safeApprove(collateralVault, assets);
    IBaseVault(collateralVault).depositAndMint(assets, 0);
  }
```

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1(Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	Medium	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

3.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

Twitter
https://twitter.com/Beosin_com

Email
service@beosin.com