SMART CONTRACT AUDIT REPORT

for

GRO PROTOCOL

Prepared By: Yiqun Chen

PeckShield

June 11, 2021

Document Properties

Client	Gro Protocol
Title	Smart Contract Audit Report
Target	Gro Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 11, 2021	Xuxian Jiang	Final Release
1.0-rc1	May 24, 2021	Xuxian Jiang	Release Candidate #1
0.3	May 16, 2021	Xuxian Jiang	Additional Findings
0.2	May 14, 2021	Xuxian Jiang	Additional Findings
0.1	May 3, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

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

The Gro Protocol effectively tokenizes stable coin investments and segments the yield and risk into two assets: one with leverage (GVT) and one with insurance (PWRD). These two exist in a relation to one another based on their ratio. Yield from the PWRD gets transferred to the GVT based on the utilization ratio of the two tokens, creating an incentive for the GVT side to take on the risk of the PWRD side. On the flip side, and stable coin or protocol failure will first and foremost be paid out from the GVT side, thus preventing PWRD holder from losing any assets in the event of an exploit or other issue.

The basic information of the Gro Protocol is as follows:

Table 1.1: Basic Information of Gro Protocol

Item	Description
Issuer	Gro Protocol
Website	https://app.gro.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 11, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories (e.g., `insurance`, `pnl`, and `pools`) and this audit does not cover the `vaults` as well as the associated `strategies` sub-directories.

- https://github.com/groLabs/gro-protocol.git (aaf7ced)

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

- https://github.com/groLabs/gro-protocol.git (f27658e)

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.

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), 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 1.3: 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the Gro 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	2	■■
Informational	2	■■
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, 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 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings of Gro Protocol

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Logic In ChainPrice::addAggregators()	Business Logic	Fixed
PVE-002	Medium	Permissionless Privileged Functions in LifeGuard3Pool	Security Features	Fixed
PVE-003	Low	Possible Front-Running/MEV For Reduced Returns	Time and State	Confirmed
PVE-004	Medium	Accommodation of Non-ERC20-Compliant Token Contracts	Business Logic	Fixed
PVE-005	Low	Proper dollarAmount Calculation in LifeGuard3Pool::invest()	Business Logic	Fixed
PVE-006	Informational	Redundant Code Removal	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Logic In ChainPrice::addAggregators()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ChainPrice
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Gro protocol has an core Buoy3Pool contract that acts as the price oracle to calculate prices of underlying assets and LP tokens in Curve pool. It also provides functionality for changing between stable coins, LP tokens and USD values. In the meantime, it also provides functionality to health check the curve pools pricing. In essence, it validates the price by comparing price ratios between the assets in the Curve pool with price ratios of an external oracle.

In the following, we show the addAggregators() routine from the external oracle ChainPrice. It comes to our attention that the given tokenIndex parameter can be better validated by ensuring require(tokenIndex < tokens.length), instead of current require(tokenIndex <= tokens.length (line 44).

```
40     function addAggregators(uint256 tokenIndex, address _aggregator)
41         external
42         onlyGovernance
43     {
44         require(tokenIndex <= tokens.length, 'invalid token index');
45         require(_aggregator != address(0), 'Invalid aggregator address');
46         address _token = tokens[tokenIndex];
47         if (tokenPriceFeed[_token].latestPrice != 0) {
48             delete tokenPriceFeed[_token];
49         }
50         tokenPriceFeed[_token].aggregator = AggregatorV3Interface(_aggregator);
51         tokenPriceFeed[_token].decimals = uint(10)**IERC20Detailed(_token).decimals();
52         _updatePriceFeed(_token);
```

```
        emit LogNewEthStableTokenAggregator(_token, _aggregator);
    }
```

Listing 3.1: ChainPrice::addAggregators()

Recommendation Revise the above `addAggregators()` routine to better validate the given input arguments.

Status The issue has been fixed by this commit: `c2fdb5c`.

3.2 Permissionless Privileged Functions in LifeGuard3Pool

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `LifeGuard3Pool`
- Category: Security Features [5]
- CWE subcategory: CWE-282 [1]

Description

The `Gro` protocol has another core `Lifeguard` contract that is designed to rebalance the investment according to the target distributions the system needs to meet in order to guarantee insurance. It does so by swapping assets through the `Curve 3pool`. Any deposited stable coins that could potentially affect the exposure negatively will be swapped to a more favorable coin by `Lifeguard`, and a dollar value associated with the deposit will be returned in order to establish the number of tokens to mint. In a similar fashion, the `Lifeguard` will act as a conduit to allow the protocol to withdraw overexposed stable coins, and swap the withdrawn coin to what users wishes to withdraw from the protocol.

```
    /// @notice Set the upper limit to the amount of assets the lifeguard will
    ///     hold on to before signaling that an invest to Curve action is necessary.
    /// @param _investToCurveThreshold New invest threshold
    function setInvestToCurveThreshold(uint256 _investToCurveThreshold) external {
        investToCurveThreshold = _investToCurveThreshold;
        emit LogNewCurveThreshold(_investToCurveThreshold);
    }

    /// @notice Set lifeguard to check Curve against external oracle
    /// @param check Check / no check
    function setHealthCheck(bool check) external {
        healthCheck = check;
        emit LogHealhCheckUpdate(check);
    }
```

Listing 3.2: LifeGuard3Pool::setInvestToCurveThreshold()

During our analysis of this Lifeguard contract, we notice it has two privileged functions that are unfortunately permission-less. To elaborate, we show above these two routines. Note these two routines manages two related protocol-wide risk parameters investToCurveThreshold and healthCheck. And the latter can be exploited to subvert the need of validating price ratios between the assets in the Curve 3pool with price ratios of an external oracle.

Recommendation Validate the caller of the above two privileged functions.

Status The issue has been fixed by this commit: 07171ec.

3.3 Possible Front-Running/MEV For Reduced Returns

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: LifeGuard3Pool
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

Within the Gro protocol, there is a constant need of swapping one stable coin to another and interacting with the Curve 3pool by adding or removing liquidity. To elaborate, we show below an example routine, i.e., investToCurveVault().

As the name indicates, this routine performs the intended investment with current asset balance by adding them as liquidity into the Curve 3pool (line 121).

```
115     function investToCurveVault() external override onlyWhitelist {
116         uint256[N_COINS] memory _inAmounts;
117         for (uint256 i = 0; i < N_COINS; i++) {
118             _inAmounts[i] = assets[i];
119             assets[i] = 0;
120         }
121         crv3pool.add_liquidity(_inAmounts, 0);
122         _investToVault(3, false);
123     }
```

Listing 3.3: LifeGuard3Pool::investToCurveVault()

We notice the liquidity addition is routed to Curve 3pool without specifying any restriction on the returned liquidity amount. As a result, it is susceptible to possible front-running/MEV attacks, resulting in a smaller gain for this round of liquidity operation.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

We emphasize that a number of other related functions also operate with the `Curve 3pool`. Examples include `distributeCurveVault()`, `depositStable()`, and `distributeCurveVault()`. The same concern is also applicable to them.

Recommendation Develop an effective mitigation (e.g., slippage control) to better protect the interests of investing users.

Status This issue has been confirmed. And the team has assured that this has been considered when evaluating the usage of `Curve` for internal swapping and deposit actions

3.4 Accommodation of Non-ERC20-Compliant Token Contracts

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: `LifeGuard3Pool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
```

```
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `withdrawSingleCoin()` routine in the `LifeGuard3Pool` contract. Since the `USDT` token is supported as `coin`, the unsafe version of `coin.transfer(recipient, balance);` (line 284) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!


```
249     function withdrawSingleCoin(
250         uint256 inAmount,
251         uint256 i,
252         uint256 minAmount,
253         address recipient
254     ) external override onlyWhitelist returns (uint256 usdAmount, uint256 balance) {
255         if (healthCheck) require(_poolCheck(), "withdrawSingle: !pool unhealthy");
256         IERC20 coin = IERC20(buoy.tokens(i));
257         balance = coin.balanceOf(address(this)).sub(assets[i]);
258         // Are available assets - locked assets for LP vault more than required
259         // minAmount. Then estimate USD value and transfer...
260         if (minAmount <= balance) {
261             uint256[] memory inAmounts = new uint256[](N_COINS);
262             inAmounts[i] = balance;
263             usdAmount = buoy.stableToUsd(inAmounts, false);
264         // ...if not, swap other loose assets into target assets before
265         // estimating USD value and transfering.
266         } else {
267             for (uint256 j; j < N_COINS; j++) {
```

```
                if (j == i) continue;
                IERC20 inCoin = IERC20(buoy.tokens(j));
                uint256 inBalance = inCoin.balanceOf(address(this)).sub(assets[j]);
                if (inBalance > 0) {
                    _exchange(inBalance, int128(j), int128(i));
                    if (coin.balanceOf(address(this)).sub(assets[i]) >= minAmount) {
                        break;
                    }
                }
            }
            balance = coin.balanceOf(address(this)).sub(assets[i]);
            uint256[] memory inAmounts = new uint256[](N_COINS);
            inAmounts[i] = balance;
            usdAmount = buoy.stableToUsd(inAmounts, false);
        }
        require(balance >= minAmount);
        coin.transfer(recipient, balance);
    }
```

Listing 3.5: LifeGuard3Pool::withdrawSingleCoin()

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been fixed by this commit: `1b06a68`.

3.5 Proper dollarAmount Calculation in LifeGuard3Pool::invest()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LifeGuard3Pool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.2, the `Lifeguard` contract is designed to rebalance the investment according to the target distributions the system needs to meet in order to guarantee insurance. It does so by swapping assets through the `Curve 3pool`. Any deposited stable coins that could potentially affect the exposure negatively will be swapped to a more favorable coin by `Lifeguard`, and a dollar value associated with the deposit will be returned in order to establish the number of tokens to mint. Our analysis shows the dollar value measurement logic can be improved.

To elaborate, we show below the `invest()` routine. This routine implements a rather straightforward logic in firstly withdrawing the given LP amount from the `Curve 3pool`, then transferring to the configured vaults. (Each stable coin has one associated vault.) However, the dollar amount is measured via the following statement `dollarAmount = buoy.stableToUsd(amounts, false)` (line 319), which needs to be revised as `dollarAmount = buoy.stableToUsd(amounts, true)`. The second argument indicates whether the dollar amount should be measured as `deposit` or `withdraw`. In our case, this is an intended deposit operation.

```
296     /// @notice Deposit into underlying vaults
297     /// @param depositAmount LP amount to invest
298     /// @param delta Target distribution of investment (%BP)
299     function invest(uint256 depositAmount, uint256[] calldata delta)
300         external
301         override
302         onlyWhitelist
303         returns (uint256 dollarAmount)
304     {
305         bool needSkim = true;
306         if (depositAmount == 0) {
307             depositAmount = lpToken.balanceOf(address(this));
308             needSkim = false;
309         }
310         uint256[N_COINS] memory _delta;
311         for (uint256 i; i < N_COINS; i++) {
312             _delta[i] = delta[i];
313         }
314         uint256[] memory amounts = new uint256[](N_COINS);
315         _withdrawUnbalanced(depositAmount, delta);
316         for (uint256 i = 0; i < N_COINS; i++) {
317             amounts[i] = _investToVault(i, needSkim);
318         }
319         dollarAmount = buoy.stableToUsd(amounts, false);
320         emit LogNewInvest(depositAmount, delta, amounts, dollarAmount, needSkim);
321     }
```

Listing 3.6: LifeGuard3Pool::invest()

A similar issue is also present in another routine, i.e., `Insurance::withdraw()`, during the `leftUsd` calculation.

Recommendation Revise the above affected routines to calculate the proper dollar amount.

Status The issue has been fixed by this commit: `3f805d3`.

3.6 Redundant Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

The `Gro` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Pausable`, to facilitate its code implementation and organization. For example, the `DepositHandler` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `LifeGuard3Pool::invest()` implementation, there is an internal `_delta` variable that keeps a copy of the given input argument (lines $310 - 312$). However, this internal `_delta` variable is not used anywhere.

```
299     function invest(uint256 depositAmount, uint256[] calldata delta)
300         external
301         override
302         onlyWhitelist
303         returns (uint256 dollarAmount)
304     {
305         bool needSkim = true;
306         if (depositAmount == 0) {
307             depositAmount = lpToken.balanceOf(address(this));
308             needSkim = false;
309         }
310         uint256[N_COINS] memory _delta;
311         for (uint256 i; i < N_COINS; i++) {
312             _delta[i] = delta[i];
313         }
314         uint256[] memory amounts = new uint256[](N_COINS);
315         _withdrawUnbalanced(depositAmount, delta);
316         for (uint256 i = 0; i < N_COINS; i++) {
317             amounts[i] = _investToVault(i, needSkim);
318         }
319         dollarAmount = buoy.stableToUsd(amounts, false);
320         emit LogNewInvest(depositAmount, delta, amounts, dollarAmount, needSkim);
321     }
```

Listing 3.7: LifeGuard3Pool::invest()

In the same vein, the same contract has another function `deposit()` that has an input argument `inAmounts`. But this input argument is not used either.

```
    function deposit(uint256[] calldata inAmounts)
        external
        override
        onlyWhitelist
        returns (uint256 newAssets)
    {
        if (healthCheck) require(_poolCheck(), "deposit: !pool unhealthy");
        uint256[N_COINS] memory _inAmounts;
        for (uint256 i = 0; i < N_COINS; i++) {
            IERC20 coin = IERC20(buoy.tokens(i));
            _inAmounts[i] = coin.balanceOf(address(this)).sub(assets[i]); //skim(
                inAmounts[i], i);
        }
        uint256 previousAssets = lpToken.balanceOf(address(this));
        crv3pool.add_liquidity(_inAmounts, 0);
        newAssets = lpToken.balanceOf(address(this)).sub(previousAssets);
    }
```

Listing 3.8: LifeGuard3Pool::deposit()

Recommendation Consider the removal of the redundant code with a simplified implementation.

Status The issue has been fixed by this commit: `295b7d7`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Gro Protocol`. The audited system presents a unique addition to current DeFi offerings by effectively tokenizing stable coin investments and segmenting the associated yield and risk with leverage and insurance, respectively. The current code base is neatly organized and those identified issues are promptly confirmed and fixed.

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-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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: 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.