SMART CONTRACT AUDIT REPORT

for

GOLDROOM

Prepared By: Yiqun Chen

PeckShield

November 27, 2021

Document Properties

Client	GoldRoom
Title	Smart Contract Audit Report
Target	GoldRoom
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang, Shulin Bie, Xiaotao Wu
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 27, 2021	Xuxian Jiang	Final Release
1.0-rc	October 17, 2021	Xuxian Jiang	Release Candidate

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 source code of the GoldRoom 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 GoldRoom

GoldRoom features a fractional-algorithmic stablecoin protocol Gu with inspiration from Frax, which is open-source, permissionless, and entirely on-chain with deployment on Ethereum and other chains. The end goal of the Gu stablecoin is to provide a highly scalable, decentralized, algorithmic money in place of fixed-supply digital assets. GoldRoom also features a number of extensions and integrations with staking and governance to facilitate the evolution of the entire ecosystem.

The basic information of GoldRoom is as follows:

Table 1.1: Basic Information of GoldRoom

Item	Description
Issuer	GoldRoom
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 27, 2021

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

- https://github.com/GoldRoomProject/gold_room.git (451bece)

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

- https://github.com/GoldRoomProject/gold_room.git (0858e36)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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

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

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) [7], 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 implementation of the `GoldRoom` 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	3	■■■
Low	5	■■■■■
Informational	0	
Total	8	

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 5 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of GoldRoom Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Logic in LockRewards::withdrawLocked()	Business Logic	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-003	Medium	Timely updateReward() Upon the rewardRate Change	Business Logic	Fixed
PVE-004	Low	Inconsistency Between GuStableCoin And Xau	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Business Logic	Mitigated
PVE-006	Low	Incorrect Trading Fee in UniswapV2Library	Business Logic	Fixed
PVE-007	Low	Simplified Logic of Pool::availableExcessGrDV()	Coding Practices	Fixed
PVE-008	Low	Improved Sanity Checks For System Parameters	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 Proper Logic in LockRewards::withdrawLocked()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LockRewards`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `GoldRoom` protocol has a built-in staking subsystem to incentivize staking users. If the user chooses a lockup period, the user will receive a boosted reward that is calculated from the chosen lockup period. While examining the boosted logic for staking and unstaking, we notice the logic needs to be improved.

Specifically, the issue stems from the lack of boosted amount calculation when the staked assets are removed. To elaborate, we show below the related `withdrawLocked()` function. As the name indicates, this function is designed to withdraw locked funds that were staked before. And the withdraw logic needs to properly reduct the withdrawn amount from the recorded staking token balance and boosted balance. However, the reduction from the boosted balance (line 187) fails to take into consideration of lockup period. Moreover, it decreases the boosted supply with the wrong amount (line 191) and burns the wrong number of `veTokens` (line 200).

```
169     function withdrawLocked(bytes32 kek_id) external override nonReentrant {
170         LockedStake memory thisStake;
171         thisStake.amount = 0;
172         uint theIndex;
173         for (uint i = 0; i < lockedStakes[msg.sender].length; i++){
174             if (kek_id == lockedStakes[msg.sender][i].kek_id){
175                 thisStake = lockedStakes[msg.sender][i];
176                 theIndex = i;
177                 break;
178             }
179         }
```

```
        require(thisStake.kek_id == kek_id, "Stake not found");
        require(block.timestamp >= thisStake.ending_timestamp  unlockedStakes == true, "Stake is still locked!");

        uint256 theAmount = thisStake.amount;
        if (theAmount > 0){
            // Staking token balance and boosted balance
            _locked_balances[msg.sender] = _locked_balances[msg.sender].sub(theAmount);
            _boosted_balances[msg.sender] = _boosted_balances[msg.sender].sub(theAmount);

            // Staking token supply and boosted supply
            _staking_token_supply = _staking_token_supply.sub(theAmount);
            _staking_token_boosted_supply = _staking_token_boosted_supply.sub(theAmount);

            // Remove the stake from the array
            delete lockedStakes[msg.sender][theIndex];

            // Give the tokens to the withdrawer
            stakingToken.safeTransfer(msg.sender, theAmount);

            // burn the veToken corresponding to Token
            VEToken(veToken).burn(msg.sender, theAmount);

            emit WithdrawnLocked(msg.sender, theAmount, kek_id);
        }

    }
```

Listing 3.1: `LockRewards::withdrawLocked()`

Recommendation Properly calculate the boosted amount for reduction when the locked assets are being unstaked and withdrawn.

Status This issue has been fixed in the following commit: `0858e36`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 3.2: 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 `getReward()` routine in the `StakingRewards` contract. If the `USDT` token is supported as `rewardsToken`, the unsafe version of `rewardsToken.transfer(msg.sender, reward)` (line 304) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
300    function getReward() public override nonReentrant updateReward(msg.sender) {
301        uint256 reward = rewards[msg.sender];
302        if (reward > 0) {
303            rewards[msg.sender] = 0;
304            rewardsToken.transfer(msg.sender, reward);
305            emit RewardPaid(msg.sender, reward);
306        }
307    }
```

Listing 3.3: `StakingRewards::getReward()`

Note this issue is also applicable to other routines, including `recoverERC20()` from `StakingRewards` and `TokenVesting` contracts.

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

Status This issue has been fixed in the following commit: `0858e36`.

3.3 Timely updateReward() Upon the rewardRate Change

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakingRewards`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `GoldRoom` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward rate can be dynamically configured via a specific routine `setRewardRate()`. When analyzing the specific routine, we notice the need of timely invoking `updateReward()` to update the reward distribution before the new `rewardRate` becomes effective.

```
    function setRewardRate(uint256 _new_rate) external onlyByOwnerOrGovernance {
        rewardRate = _new_rate;
    }

    function setOwnerAndTimelock(address _new_timelock) external onlyByOwnerOrGovernance {
        timelock_address = _new_timelock;
    }

    /* ========== MODIFIERS ========== */

    modifier updateReward(address account) {
        // Need to retro-adjust some things if the period hasn't been renewed, then start a new one
        if (block.timestamp > periodFinish) {
            retroCatchUp();
        }
        else {
            rewardPerTokenStored = rewardPerToken();
            lastUpdateTime = lastTimeRewardApplicable();
        }
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }
```

Listing 3.4: `StakingRewards::setRewardRate()`

If the call to `updateReward()` is not immediately invoked before updating the new `rewardRate`, the rewards may not be accrued using the right `rewardRate`. In particular, earlier time intervals may be wrongfully using the new `rewardRate`! Fortunately, this interface is restricted to the authorized entities (via the `onlyByOwnerOrGovernance` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `updateReward()` when the `rewardRate` is updated. Also, keep in mind that the current contract does not support deflationary tokens! A vetting process needs to be in place to ensure incompatible deflationary tokens will not be supported as the staking token for reward.

Status This issue has been fixed in the following commit: `0858e36`.

3.4 Inconsistency Between GuStableCoin And Xau

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GuStableCoin`, `Xau`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `GoldRoom` protocol has two built-in fractional-algorithmic stablecoins `Gu` and `Xau`. Our analysis shows that these two stablecoins share the same logic, but have unnecessary inconsistency in their implementation.

To elaborate, we show below the inconsistent implementation of the `pool_burn_from()` function. This function is called by pools when user redeems the stablecoins. It comes to our attention that the `Gu` version directly burns the given amount from the user via the underlying routine `_burn()`. However, the `Xau` version achieves the same purpose by calling another underlying routine `_burnFrom()`. The `Xau` version requires the user to approve the spending authorization, which is apparently different from the `Gu` counterpart.

```
    // Used by pools when user redeems
    function pool_burn_from(address b_address, uint256 b_amount) public onlyPools {
        super._burn(b_address, b_amount);
        emit GuBurned(b_address, msg.sender, b_amount);
    }
```

Listing 3.5: `GuStablecoin::pool_burn_from()`

```
    // Used by pools when user redeems
    function pool_burn_from(address b_address, uint256 b_amount) public onlyPools {
        super._burnFrom(b_address, b_amount);
```

```
        emit XauBurned(b_address, msg.sender, b_amount);
    }
```

Listing 3.6: `Xau::pool_burn_from()`

Recommendation Be consistent in the above two stablecoins `Gu` and `Xau`.

Status This issue has been fixed in the following commit: `0858e36`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `GoldRoom` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and oracle adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setTimelock(address new_timelock) external onlyByOwnGov {
        require(new_timelock != address(0), "Timelock address cannot be 0");
        timelock_address = new_timelock;
    }

    function setSynthAddress(address Synth_contract_address) external onlyByOwnGov {
        require(Synth_contract_address != address(0), "Zero address detected");

        Synth = ISynth(Synth_contract_address);

        emit SynthAddressSet(Synth_contract_address);
    }
```

Listing 3.7: Example `Setters` in the `GrShares` Contract

Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 confirmed with the team. For the time being, it will be mitigated by a 24-hour timelock to balance efficiency and timely adjustment. After the protocol becomes stable, it is expected to migrate to a multi-sig account, and eventually be managed by community proposals for decentralized governance.

3.6 Incorrect Trading Fee in UniswapV2Library

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `GoldRoom` protocol, a number of situations require the real-time swap of one token to another. For example, the library function `getAmountOut()` computes the output token amount when given an input amount and respective pair reserves. Since the `Gu` stablecoin is intended for `BSC` deployment, it is important to keep in mind that the default DEX used in `Gu` is `PancakeSwap`. If you make a token swap or trade on the exchange, you will need to pay a 0.25% trading fee, which is broken down into two parts. The first part of 0.17% is returned to liquidity pools in the form of a fee reward for liquidity providers, the 0.03% is sent to the `PancakeSwap Treasury`, and the remaining 0.05% is used towards `CAKE` buyback and burn.

However, if we examine the library contract `UniswapV2Library`, it has implicitly assumed the trading fee is 0.3%, instead of 0.25%. The difference in the built-in trading fee with the actual `PancakeSwap` may skew the token conversion calculation.

```
    // given an input amount of an asset and pair reserves, returns the maximum output
        amount of the other asset
    function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal pure
        returns (uint amountOut) {
        require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
```

```
        require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library:
            INSUFFICIENT_LIQUIDITY');
        uint amountInWithFee = amountIn.mul(997);
        uint numerator = amountInWithFee.mul(reserveOut);
        uint denominator = reserveIn.mul(1000).add(amountInWithFee);
        amountOut = numerator / denominator;
    }

    // given an output amount of an asset and pair reserves, returns a required input
        amount of the other asset
    function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) internal pure
        returns (uint amountIn) {
        require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library:
            INSUFFICIENT_LIQUIDITY');
        uint numerator = reserveIn.mul(amountOut).mul(1000);
        uint denominator = reserveOut.sub(amountOut).mul(997);
        amountIn = (numerator / denominator).add(1);
    }
```

Listing 3.8: UniswapV2Library::getAmountOut()/getAmountIn()

Recommendation Make the built-in trading fee consistent with the actual trading fee in `PancakeSwap`.

Status This issue has been fixed in the following commit: `0858e36`.

3.7 Simplified Logic of Pool::availableExcessGrDV()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `Pool::availableExcessGrDV()` as an example. This routine is used to return the value of excess `Gr` held in the `Synth` pool, compared to what is needed to maintain the global collateral ratio.

```
203    function availableExcessGrDV() public view returns (uint256) {

205        uint256 synth_total_supply = Synth.totalSupply();
206        uint256 global_collateral_ratio = Synth.global_collateral_ratio();
207        uint256 global_collat_value = Synth.globalCollateralValue();

209        uint256 effective_collateral_ratio = global_collat_value.mul(1e6).div(
               synth_total_supply); //returns it in 1e6
210        if (global_collateral_ratio.mul(synth_total_supply) > synth_total_supply.mul(
               effective_collateral_ratio)){
211            return (global_collateral_ratio.mul(synth_total_supply).sub(
                   synth_total_supply.mul(effective_collateral_ratio))).div(1e6);
212        }
213        return 0;
214    }
```

Listing 3.9: Pool::availableExcessGrDV()

We notice the comparison calculation (line 210) can be simplified as `if (global_collateral_ratio > effective_collateral_ratio)`. In other words, there is no need to multiple `synth_total_supply` in both sides without any precision loss.

Recommendation Simplify the above calculation without precision loss.

Status This issue has been fixed in the following commit: `0858e36`.

3.8 Improved Sanity Checks For System Parameters

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `GoldRoom` protocol is no exception. Specifically, if we examine the `Pool` contract, it has defined a number of protocol-wide risk parameters, e.g., `pool_ceiling`, `bonus_rate`, `minting_fee`, and `recollat_fee`. In the following, we show an example routine that allows for their changes.

```
537    // Combined into one function due to 24KiB contract memory limit
```

```
    function setPoolParameters(uint256 new_ceiling, uint256 new_bonus_rate, uint256
        new_redemption_delay, uint256 new_mint_fee, uint256 new_redeem_fee, uint256
        new_buyback_fee, uint256 new_recollat_fee) external onlyByOwnGov {
        pool_ceiling = new_ceiling;
        bonus_rate = new_bonus_rate;
        redemption_delay = new_redemption_delay;
        minting_fee = new_mint_fee;
        redemption_fee = new_redeem_fee;
        buyback_fee = new_buyback_fee;
        recollat_fee = new_recollat_fee;

        emit PoolParametersSet(new_ceiling, new_bonus_rate, new_redemption_delay,
            new_mint_fee, new_redeem_fee, new_buyback_fee, new_recollat_fee);
    }

    function setPriceThresholds(uint256 new_mint_price_threshold, uint256
        new_redeem_price_threshold) external onlyByOwnGov {
        mint_price_threshold = new_mint_price_threshold;
        redeem_price_threshold = new_redeem_price_threshold;
        emit PriceThresholdsSet(new_mint_price_threshold, new_redeem_price_threshold);
    }

    function setTimelock(address new_timelock) external onlyByOwnGov {
        timelock_address = new_timelock;

        emit TimelockSet(new_timelock);
    }
```

Listing 3.10: Examples `Setters` in `Pool`

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large `mining_fee` parameter will make every minting operation extremely expensive.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status This issue has been fixed in the following commit: `0858e36`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `GoldRoom` protocol that has a built-in `Gu`, which is a fractional-algorithmic stablecoin protocol with inspiration from `Frax`. The audited `GoldRoom` protocol also features a number of extensions and integrations for staking and governance to facilitate the evolution of the entire ecosystem. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.