SMART CONTRACT AUDIT REPORT

for

GemKeeper Protocol

Prepared By: Yiqun Chen

PeckShield

February 18, 2022

Document Properties

Client	GemKeeper
Title	Smart Contract Audit Report
Target	GemKeeper
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 18, 2022	Xuxian Jiang	Final Release
1.0-rc1	February 12, 2022	Xuxian Jiang	Release Candidate #1

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 GemKeeper 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 GemKeeper

GemKeeper is a one-stop DeFi platform on the Oasis Emerald paratime. The first product released under the GemKeeper umbrella is an Automated Market Maker style DEX. The DEX supports standard features such as swapping tokens, creating liquidity pools, adding/removing tokens from liquidity pools, and staking GemKeeper Liquidity Pool tokens (GLP tokens) to earn GemKeeper's BLING token rewards. Swaps made on the GemKeeper DEX pay a 0.3% fee which will go to the liquidity providers. In the future, swap fees will be split so that 0.25% goes to the liquidity providers and the other 0.05% revenue is shared with Bling token holders who are staked in the protocol. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of GemKeeper

Item	Description
Name	GemKeeper
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 18, 2022

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

- https://github.com/GemKeeperDEV/GemKeeperFinance.git (8712e70)

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

- https://github.com/GemKeeperDEV/GemKeeperFinance.git (5551457)

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 the 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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 checklist of 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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

Table 1.4: 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 Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `GemKeeper` 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 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	1	■
Low	4	■ ■ ■ ■
Undetermined	1	■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, 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 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 1 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 undetermined-severity issue.

Table 2.1: Key GemKeeper Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Fork-Compliant Domain Separator in UniswapV2ERC20	Business Logic	Confirmed
PVE-002	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Resolved
PVE-003	Low	Duplicate Pool Detection and Prevention	Business Logic	Confirmed
PVE-004	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Resolved
PVE-005	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	Resolved
PVE-006	Low	Reentrancy Risk in MasterChef	Time and State	Resolved

Besides 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 Fork-Compliant Domain Separator in UniswapV2ERC20

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `UniswapV2ERC20`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `UniswapV2ERC20` token contract strictly follows the widely-accepted ERC20 specification. In the meantime, we notice the support of EIP-2612 with the `permit()` function that allows for approvals to be made via `secp256k1` signatures. Interestingly, we notice the state variable `DOMAIN_SEPARATOR` is initialized once inside the `constructor()` function (lines 30-38).

```
25    constructor() public {
26        uint chainId;
27        assembly {
28            chainId := chainid()
29        }
30        DOMAIN_SEPARATOR = keccak256(
31            abi.encode(
32                keccak256('EIP712Domain(string name,string version,uint256 chainId,
                    address verifyingContract)'),
33                keccak256(bytes(name)),
34                keccak256(bytes('1')),
35                chainId,
36                address(this)
37            )
38        );
39    }
```

Listing 3.1: `UniswapV2ERC20::constructor()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function and should be unique to the contract and the chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed `DOMAIN_SEPARATOR`, a valid signature for one chain could be replayed on the other.

```
function permit(address owner, address spender, uint value, uint deadline, uint8 v,
    bytes32 r, bytes32 s) external {
    require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
    bytes32 digest = keccak256(
        abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[
                owner]++, deadline))
        )
    );
    address recoveredAddress = ecrecover(digest, v, r, s);
    require(recoveredAddress != address(0) && recoveredAddress == owner, 'UniswapV2:
         INVALID_SIGNATURE');
    _approve(owner, spender, value);
}
```

Listing 3.2: `UniswapV2ERC20::permit()`

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` inside the `permit()` function.

Status The issue has been confirmed.

3.2 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UniswapV2Router02`
- Category: Coding Practices [4]
- CWE subcategory: CWE-628 [1]

Description

In the `UniswapV2Router02` contract, the `addLiquidity()` routine (see the code snippet below) is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity via the `UniswapRouterV2::addLiquidity()` routine. To elaborate, we show below the related code snippet.

```
function addLiquidity(
    address tokenA,
    address tokenB,
```

```
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint amountA, uint amountB,
        uint liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IUniswapV2Pair(pair).mint(to);
    }
```

Listing 3.3: `UniswapV2Router02::addLiquidity()`

```
    // **** ADD LIQUIDITY ****
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory, tokenA,
            tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'UniswapV2Router:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
```

```
    }
```

Listing 3.4: UniswapV2Router02::_addLiquidity()

It comes to our attention that the UniswapV2 Router has implicit assumptions on the _addLiquidity() routine. The above routine takes two amounts: amountXDesired and amountXMin. The first amount amountXDesired determines the desired amount for adding liquidity to the pool and the second amount amountXMin determines the minimum amount of used assets. There are two implicit conditions, i.e., amountADesired >= amountAMin and amountBDesired >= amountBMin. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on UniswapV2 Router may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of amountADesired >= amountAMin and amountBDesired >= amountBMin explicitly in the addLiquidity() function.

Status This issue has been fixed in the following commit: c9b9c42.

3.3 Duplicate Pool Detection and Prevention

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The GemKeeper protocol provides a built-in incentive mechanism that rewards the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a privileged function). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in add(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

```
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accBlingPerShare: 0
            })
        );
    }
```

Listing 3.5: `MasterChef::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
```

```
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accBlingPerShare: 0
            })
        );
    }
```

Listing 3.6: `MasterChef::add()`

Status This issue has been confirmed and the team clarifies extra care will be exercised to avoid the addition of a duplicate pool.

3.4 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `MasterChef` protocol provides an incentive mechanism that rewards 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 pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's BLING allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
```

```
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.7: `MasterChef::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    // Update the given pool's BLING allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.8: Revised `MasterChef::set()`

Status This issue has been fixed in the following commit: 2a1c1b5.

3.5 Staking Incompatibility With Deflationary Tokens

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `MasterChef`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the `GemKeeper` protocol, the `MasterChef` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
                user.amount.mul(pool.accBlingPerShare).div(1e12).sub(
                    user.rewardDebt
                );
            if (block.timestamp >= tokenLaunchTime  tokenLaunched) {
                pending = pending.add(user.unclaimedReward);
                user.unclaimedReward = 0;
                safeBlingTransfer(msg.sender, pending);
            } else {
                user.unclaimedReward = user.unclaimedReward.add(pending);
            }

        }
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accBlingPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.9: `MasterChef::deposit())`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accBlingPerShare` via dividing `blingReward` by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 209). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accBlingPerShare` as the final result, which dramatically inflates the pool's reward.

```
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 blingReward =
            multiplier.mul(blingPerBlock).mul(pool.allocPoint).div(
                totalAllocPoint
            );
        bling.mint(devaddr, blingReward.mul(10).div(65));
        bling.mint(address(this), blingReward);
        pool.accBlingPerShare = pool.accBlingPerShare.add(
            blingReward.mul(1e12).div(lpSupply)
        );
        pool.lastRewardBlock = block.number;
    }
```

Listing 3.10: `MasterChef::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `GemKeeper` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed and the team clarifies no deflationary tokens will be supported.

3.6 Reentrancy Risk in MasterChef

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 285) starts before effecting the update on the internal state (line 287), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
281     // Withdraw without caring about rewards. EMERGENCY ONLY.
282     function emergencyWithdraw(uint256 _pid) public {
283         PoolInfo storage pool = poolInfo[_pid];
284         UserInfo storage user = userInfo[_pid][msg.sender];
285         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
286         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
287         user.amount = 0;
288         user.rewardDebt = 0;
289     }
```

Listing 3.11: `MasterChef::emergencyWithdraw()`

Note that other routines share the same issue, including `deposit()`, `withdraw()`, and `emergencyWithdraw()`, from the same contract.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in the following commit: f454dcb.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `GemKeeper` protocol, which is a one-stop DeFi platform on the `Oasis Emerald` paratime. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.