Token & MasterChef

Smart Contract Audit Report
Prepared for SamoyedFinance

Date Issued:	Sep 21, 2021
Project ID:	AUDIT2021020
Version:	v1.0
Confidentiality Level:	Public

inspex
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021020
Version	v1.0
Client	SamoyedFinance
Project	Token & MasterChef
Auditor(s)	Suvicha Buakhom Patipon Suwanbol
Author	Suvicha Buakhom
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Sep 21, 2021	Full report	Suvicha Buakhom

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary 1
1.1. Audit Result 1
1.2. Disclaimer 1
2. Project Overview 2
2.1. Project Introduction 2
2.2. Scope 3
3. Methodology 4
3.1. Test Categories 4
3.2. Audit Items 5
3.3. Risk Rating 6
4. Summary of Findings 7
5. Detailed Findings Information 9
5.1. Improper Delegation Handling 9
5.2. Improper Reward Calculation (Duplicated LP Token) 13
5.3. Improper Reward Calculation (BONUS_MULTIPLIER) 16
5.4. Improper Reward Calculation (smoyPerBlock) 19
5.5. Improper Reward Calculation (_withUpdate) 21
5.6. Centralized Control of State Variable 25
5.7. Design Flaw in massUpdatePools() Function 27
5.8. Unchecked Deposit Fee Value 29
5.9. Insufficient Logging for Privileged Functions 34
5.10. Unsupported Design for Deflationary Token 36
5.11. Improper Function Visibility 41
6. Appendix 43
6.1. About Inspex 43
6.2. References 44

1. Executive Summary

As requested by SamoyedFinance, Inspex team conducted an audit to verify the security posture of the Token & MasterChef smart contracts between Sep 7, 2021 and Sep 8, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Token & MasterChef smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 high, 5 medium, 2 low, 1 very low, and 2 info-severity issues. With the project team's prompt response, 1 high, 5 medium, 2 low, 1 very low, and 1 info-severity issues were resolved or mitigated in the reassessment, while 1 info-severity issue was acknowledged by the team. Therefore, Inspex trusts that Token & MasterChef smart contracts have sufficient protections to be safe for public use.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

SamoyedFinance is an NFT lottery, gaming and yield farming platform which gives the users unlimited chances to win prizes. The users can win never-ending jackpots, play games, and exchange NFT all in one platform.

Token & MasterChef smart contracts are the main contracts responsible for distributing $SMOY on the platform. The users can deposit tokens to the pools added to the MasterChef contract and earn $SMOY as a reward.

Scope Information:

Project Name	Token & MasterChef
Website	https://samoyedfinance.app/farm
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Sep 7, 2021 - Sep 8, 2021
Reassessment Date	Sep 20, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit:

Contract	Location (URL)
SamoyToken	https://testnet.bscscan.com/address/0x124B4c3d31aEe6C05176ab18478E3C167b078618#code
SamoyedMasterChef	https://testnet.bscscan.com/address/0xA650430d3daA62740C3Dd3c0a6f88C394f57a93d#code
KennelClub	https://testnet.bscscan.com/address/0xb929c08768b507582545303107c7791b67f4db00#code

Reassessment:

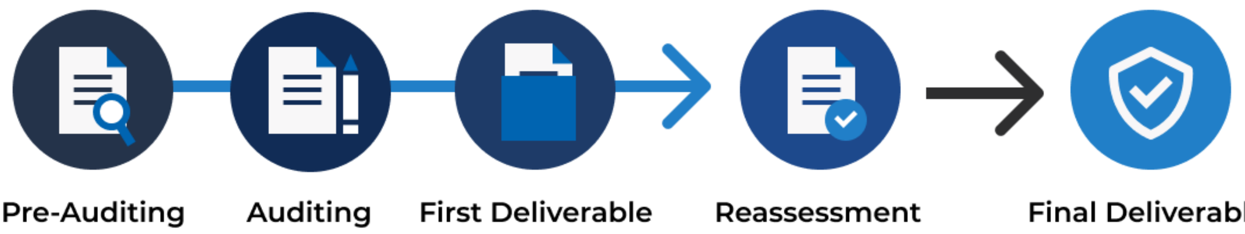
Contract	Location (URL)
SamoyToken	https://bscscan.com/address/0xBdb44DF0A914c290DFD84c1eaf5899d285717fdc#code
SamoyedMasterChef	https://bscscan.com/address/0x5D21D02378670119453530478288AEe67b807e2a#code
KennelClub	https://bscscan.com/address/0x1364e039de60522aef045095823148e5e20f649a#code

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Use of Upgradable Contract Design
Insufficient Logging for Privileged Functions
Improper Kill-Switch Mechanism
Improper Front-end Integration

Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

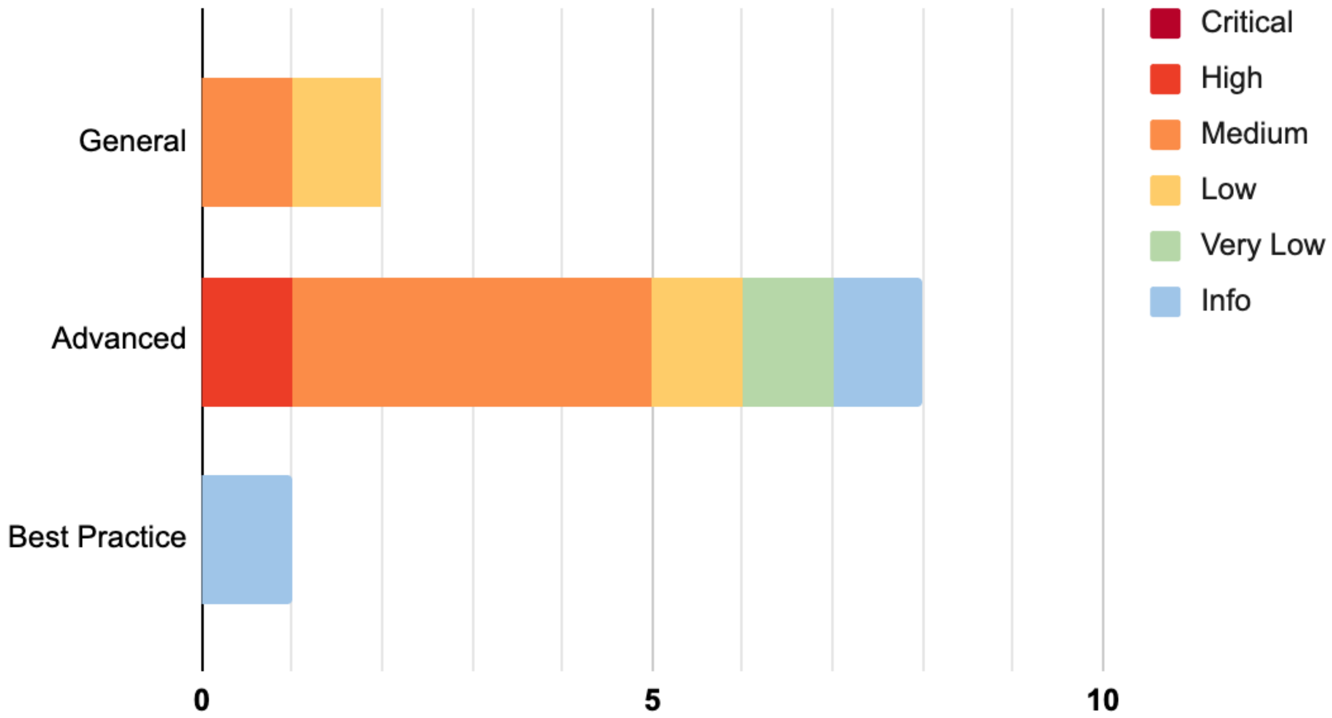
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood / Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

4. Summary of Findings

From the assessments, Inspex has found 11 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Improper Delegation Handling	Advanced	High	Resolved
IDX-002	Improper Reward Calculation (Duplicated LP Token)	Advanced	Medium	Resolved
IDX-003	Improper Reward Calculation (BONUS_MULTIPLIER)	Advanced	Medium	Resolved
IDX-004	Improper Reward Calculation (smoyPerBlock)	Advanced	Medium	Resolved
IDX-005	Improper Reward Calculation (_withUpdate)	Advanced	Medium	Resolved
IDX-006	Centralized Control of State Variable	General	Medium	Resolved
IDX-007	Design Flaw in massUpdatePools() Function	General	Low	Resolved *
IDX-008	Unlimit Deposit Fee	Advanced	Low	Resolved
IDX-009	Insufficient Logging for Privileged Functions	Advanced	Very Low	Resolved
IDX-010	Unsupported Design for Deflationary Token	Advanced	Info	Resolved
IDX-011	Improper Function Visibility	Best Practice	Info	No Security Impact

* The mitigations or clarifications by SamoyedFinance can be found in Chapter 5.

5. Detailed Findings Information

5.1. Improper Delegation Handling

ID	IDX-001
Target	KennelClub
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium The number of votes can be manipulated, causing the result of voting to be unfair and untrustworthy. **Likelihood:** High Manipulating the vote result gives advantages to the abusers. This motivates anyone to exploit this scenario since there is no mechanism to prevent it.
Status	Resolved SamoyedFinance team has resolved this issue by removing the delegation feature in the deployed contract on mainnet. `KennelClub` contract: https://bscscan.com/address/0x1364e039de60522aef045095823148e5e20f649a

5.1.1. Description

In the `KennelClub` contract, there is a voting mechanism implemented, allowing the users (Delegators) to delegate their votes to another address (Delegatees) without transferring their tokens.

The users can delegate their votes to another address using the `delegate()` function, which calls the `_delegate()` function.

KennelClub.sol

```
function delegate(address delegatee) external {
    return _delegate(msg.sender, delegatee);
}
```

The `_delegate()` function sets the delegatee of the address in line 177, and transfer the number of votes from the old delegatee to the new delegatee with the current token balance of the delegator by using the `_moveDelegates()` function as in line 181.

KennelClub.sol

```
function _delegate(address delegator, address delegatee) internal {
    address currentDelegate = _delegates[delegator];
    uint256 delegatorBalance = balanceOf(delegator); // balance of underlying CAKEs (not scaled);
    _delegates[delegator] = delegatee;

    emit DelegateChanged(delegator, currentDelegate, delegatee);

    _moveDelegates(currentDelegate, delegatee, delegatorBalance);
}
```

The `_moveDelegates()` function calculates the new amounts of votes for the delegatees.

KennelClub.sol

```
function _moveDelegates(
    address srcRep,
    address dstRep,
    uint256 amount
) internal {
    if (srcRep != dstRep && amount > 0) {
        if (srcRep != address(0)) {
            // decrease old representative
            uint32 srcRepNum = numCheckpoints[srcRep];
            uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
            uint256 srcRepNew = srcRepOld.sub(amount);
            _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
        }

        if (dstRep != address(0)) {
            // increase new representative
            uint32 dstRepNum = numCheckpoints[dstRep];
            uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
            uint256 dstRepNew = dstRepOld.add(amount);
            _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
        }
    }
}
```

However, the delegate mechanism will only activate when the delegator calls the `delegate()` function. This means $KENNEL could be transferred to another person after the first delegation, and that person can call the `delegate()` function again, allowing $KENNEL to be used for double spending attack in an aspect of voting mechanism, for example:

The total of $KENNEL and votes of each user in this scenario is represented in the table below:

User	$KENNEL	Vote
A	100	100
B	0	0
C	0	0

User A delegates 100 votes to User B, now User B has 100 votes.

User	$KENNEL	Vote
A	100	0
B	0	100
C	0	0

User A transfers 100 $KENNEL to User C, now User C has 100 $KENNEL

User	$KENNEL	Vote
A	0	0
B	0	100
C	100	100

User C delegates 100 Votes to User B, now Use B has 200 votes.

User	$KENNEL	Vote
A	0	0
B	0	200
C	100	0

This process can be done repeatedly to increase the voting power without any limit.

5.1.2. Remediation

Inspex suggests that the delegation vote should be transferred from the previous delegatee to the new delegatee when the token transfer occurs.

Since the `KennelClub` contract is implemented by following the ERC20 standard, inserting the `_moveDelegates` function to the `transfer()` and `transferFrom()` functions will solve this issue.

KennelClub.sol

```
function transfer(address recipient, uint256 amount) external override nonReentrant returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    _moveDelegates(_delegates[msgSender()], _delegates[recipient], amount);
    return true;
}

function transferFrom(
    address sender,
    address recipient,
    uint256 amount
) external override nonReentrant returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(
        sender,
        _msgSender(),
        _allowances[sender][_msgSender()].sub(amount, "BEP20: transfer amount exceeds allowance")
    );
    _moveDelegates(_delegates[sender], _delegates[recipient], amount);
    return true;
}
```

5.2. Improper Reward Calculation (Duplicated LP Token)

ID	IDX-002
Target	SamoyedMasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $SMOY reward miscalculation can lead to an unfair $SMOY token distribution to the users. **Likelihood:** Medium It is possible that the contract owner will add a new pool that uses the same token as other pool since there is no restriction.
Status	Resolved SamoyedFinance team has resolved this issue as suggested in the deployed contract on mainnet. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D02378670119453530478288AEe67b807e2a

5.2.1. Description

In the `SamoyedMasterChef` contract, a new staking pool can be added using the `add()` function. The staking token for the new pool is defined using the `_lpToken` variable; however, there is no additional checking whether the `_lpToken` is already used in other pools or not.

SamoyedMasterChef.sol

```
function add(
    uint256 _allocPoint,
    IBEP20 _lpToken,
    bool _withUpdate,
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external 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,
            accSmoyPerShare: 0,
            minDepositFeeRate: _minDepositFeeRate,
            maxDepositFeeRate: _maxDepositFeeRate
        })
    );
}
```

In the `updatePool()` function, the balance of `pool.lpToken` in the contract is used as a denominator to calculate `pool.accSmoyPerShare`.

SamoyedMasterChef.sol

```
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 smoyReward = multiplier.mul(smoyPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
    smoy.mintTo(devAddr, smoyReward.div(10));
    smoy.mintTo(address(kennel), smoyReward);
    pool.accSmoyPerShare = pool.accSmoyPerShare.add(smoyReward.mul(1e12).div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

When the owner of `SamoyedMasterChef` adds a pool with the same `lpToken` as another pool, the `lpToken` value is counted from all pools using the same `lpToken`, resulting in a higher value of denominator (`lpSupply`) than it should be.

5.2.2. Remediation

Inspex suggests validating the `_lpToken` address in `add()` function to prevent duplicated `_lpToken` when adding a new pool as shown in the following example:

SamoyedMasterChef.sol

```
mapping(address => bool) public isAddedPool;

function add(
    uint256 _allocPoint,
    IBEP20 _lpToken,
    bool _withUpdate,
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external onlyOwner {
    require(!isAddedPool[address(_lpToken)], "Duplicated LP Token");

    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accSmoyPerShare: 0,
            minDepositFeeRate: _minDepositFeeRate,
            maxDepositFeeRate: _maxDepositFeeRate
        })
    );

    isAddedPool[address(_lpToken)] = true;
}
```

5.3. Improper Reward Calculation (BONUS_MULTIPLIER)

ID	IDX-003
Target	SamoyedMasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $SMOY reward miscalculation can lead to an unfair $SMOY distribution to the users. **Likelihood:** Medium The bonus multiplier can only be updated by the contract owner, but it is likely that this value will be updated.
Status	Resolved SamoyedFinance team has resolved this issue as suggested in the deployed contract on mainnet. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D023786701194535304782 88AEe67b807e2a

5.3.1. Description

The `BONUS_MULTIPLIER` state variable is used as a factor to calculate the reward in `getMultiplier()` function.

SamoyedMasterChef.sol

```
function getMultiplier(uint256 _from, uint256 _to) public view returns
(uint256) {
    return _to.sub(_from).mul(BONUS_MULTIPLIER);
}
```

In order to mint the reward to the users who deposited, the `updatePool()` function is executed, calling the `getMultiplier()` function to calculate the reward, and recording the users' reward in the `accSmoyPerShare` state variable.

SamoyedMasterChef.sol

```
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 smoyReward =
multiplier.mul(smoyPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
    smoy.mintTo(devAddr, smoyReward.div(10));
    smoy.mintTo(address(kennel), smoyReward);
    pool.accSmoyPerShare =
pool.accSmoyPerShare.add(smoyReward.mul(1e12).div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

However, the `SamoyedMasterChef` contract owner can change the `BONUS_MULTIPLIER` by using the `updateMultiplier()` function.

SamoyedMasterChef.sol

```
function updateMultiplier(uint256 multiplierNumber) external onlyOwner {
    BONUS_MULTIPLIER = multiplierNumber;
}
```

Therefore, whenever the `BONUS_MULTIPLIER` variable is modified without updating the pending reward first, the reward of each pool will be incorrectly calculated.

For example:

Assuming that `BONUS_MULTIPLIER` is originally set to 1 and `smoyPerBlock` is set to 10.

Block	Action
1000000	All pools' rewards are updated.
1100000	BONUS_MULTIPLIER is updated to 5
1200000	The pools' rewards are updated once again.

The total rewards minted during block 1000000 to block 1200000 is equal to 5 * 10 $SMOY per block, from block 1000000 to block 1200000 (50 x (1200000 - 1000000) = 10,000,000 $SMOY).

However, the rewards should be calculated by accounting for the original BONUS_MULTIPLIER value during the period when it is not yet updated as follows:

- BONUS_MULTIPLIER is set to 1, from block 1000000 to block 1100000 (10 * 1 * (1100000 - 1000000) = 1,000,000 $SMOY)
- BONUS_MULTIPLIER is set to 5, from block 1100000 to block 1200000 (10 * 5 * (1200000 - 1100000) = 5,000,000 $SMOY)
- Total $SMOY minted (1,000,000 + 5,000,000 = 6,000,000 $SMOY)

5.3.2. Remediation

Inspex suggests adding `massUpdatePools()` function calling before updating `BONUS_MULTIPLIER` variable as shown in the following example:

SamoyedMasterChef.sol

```
function updateMultiplier(uint256 multiplierNumber) external onlyOwner {
    massUpdatePool();
    BONUS_MULTIPLIER = multiplierNumber;
}
```

5.4. Improper Reward Calculation (smoyPerBlock)

ID	IDX-004
Target	SamoyedMasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $SMOY reward miscalculation can lead to an unfair $SMOY token distribution to the users. **Likelihood:** Medium The `smoyPerBLock` can only be changed by the contract owner, but it is likely that this value will be updated.
Status	Resolved SamoyedFinance team has resolved this issue as suggested in the deployed contract on mainnet. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D0237867011945353O478288AEe67b807e2a

5.4.1. Description

The `smoyPerBlock` variable is used to determine the total number of $SMOY to be minted as a reward per block, so it is one of the main factors used in the rewards calculation. Therefore, whenever the `smoyPerBlock` variable is modified without updating the pending reward first, the reward of each pool will be incorrectly calculated.

In the `updateSmoyPerBlock()` function shown below, the `smoyPerBlock` variable is modified without updating the reward.

SamoyedMasterChef.sol

```
function updateSmoyPerBlock(uint256 _smoyPerBlock) external onlyOwner {
    smoyPerBlock = _smoyPerBlock;
}
```

For example:

Assuming that `smoyPerBlock` is originally set to 15 $SMOY per block.

Block	Action
1000000	All pools' rewards are updated.
1100000	smoyPerblock is updated to 20 $SMOY per block using updateSmoyPerBlock() function.
1200000	The pools' rewards are updated once again.

The total rewards minted during block 1000000 to block 1200000 is equal to 20 $SMOY per block, from block 1000000 to block 1000000 (20 x (1200000 - 1000000) = 4,000,000 $SMOY).

However, the rewards should be calculated by accounting for the original `smoyPerBlock` value during the period when it is not yet updated as follows:

- 15 $SMOY per block, from block 1000000 to block 1100000 (15 * (1100000 - 1000000) = 1,500,000 $SMOY)
- 20 $SMOY per block, from block 1100000 to block 1200000 (20 * (1200000 - 1100000) = 2,000,000 $SMOY)
- Total $SMOY minted (1,500,000 + 2,000,000 = 3,500,000 $SMOY)

5.4.2. Remediation

Inspex suggests adding `massUpdatePools()` function calling before updating the `smoyPerBlock` variable as shown in the following example:

SamoyedMasterChef.sol

```
function updateSmoyPerBlock(uint256 _smoyPerBlock) external onlyOwner {
    massUpdatePool();
    smoyPerBlock = _smoyPerBlock;
}
```

5.5. Improper Reward Calculation (_withUpdate)

ID	IDX-005
Target	SamoyedMasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $SMOY reward miscalculation can lead to an unfair $SMOY token distribution to the users. **Likelihood:** Medium The `add()` and the `set()` functions can only be called by the contract owner, but it is possible that the `totalAllocPoint` state will be changed without setting the `_withUpdate` parameter to `true`.
Status	Resolved SamoyedFinance team has resolved this issue as suggested in the deployed contract on mainnet. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D02378670119453530478288AEe67b807e2a

5.5.1. Description

The `totalAllocPoint` variable is used to determine the portion that each pool would get from the total rewards minted, so it is one of the main factors used in the rewards calculation. Therefore, whenever the `totalAllocPoint` variable is modified without updating the pending rewards first, the reward of each pool will be incorrectly calculated.

In the `add()` and `set()` functions shown below, if `_withUpdate` is set to false, the `totalAllocPoint` variable will be modified without updating the rewards (`massUpdatePools()`).

SamoyedMasterChef.sol

```
110 function add(
111     uint256 _allocPoint,
112     IBEP20 _lpToken,
113     bool _withUpdate,
114     uint256 _minDepositFeeRate,
115     uint256 _maxDepositFeeRate
116 ) external onlyOwner {
117     if (_withUpdate) {
118         massUpdatePools();
```

```
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accSmoyPerShare: 0,
            minDepositFeeRate: _minDepositFeeRate,
            maxDepositFeeRate: _maxDepositFeeRate
        })
    );
}
```

SamoyedMasterChef.sol

```
function set(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate,
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].minDepositFeeRate = _minDepositFeeRate;
    poolInfo[_pid].maxDepositFeeRate = _maxDepositFeeRate;
    if (prevAllocPoint != _allocPoint) {
        totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
    }
}
```

For example:

Assuming that on block 1000000, `smoyPerBlock` is 5 $SMOY per block, `totalAllocPoint` is 5000, and `allocPoint` of pool id 0 is 500.

Block	Action
1000000	All pools' rewards are updated
1100000	A new pool is added using the `add()` function, causing the `totalAllocPoint` to be changed

	from 5000 to 10000
1200000	The pools' rewards are updated once again.

From current logic, the total rewards allocated to the pool id 0 during block 1000000 to 1200000 is equal to 50,000 $SMOY calculated using the following equation:

Block	Total Reward Block	Total Allocation Point	Total $SMOY per block for pool 0 (smoyPerBlock*pool0allocPoint/ totalAllocPoint)	Total pool 0 $SMOY Reward
1000000 - 1200000	200000	10,000	0.25 $SMOY per block	50,000 $SMOY

However, the rewards should be calculated by accounting for the original `totalAllocPoint` value during the period when it is not yet updated as follows:

Block	Total Reward Block	Total Allocation Point	Total $SMOY per block for pool 0 (smoyPerBlock*pool0allocPoint/ totalAllocPoint)	Total pool 0 $SMOY Reward
1000000 - 1100000	100000	5,000	0.5 $SMOY per block	50,000 $SMOY
1100000 - 1200000	100000	10,000	0.25 $SMOY per block	25,000 $SMOY

The correct total $SMOY rewards is 75,000 $SMOY, which is different from the miscalculated reward by 25,000 $SMOY.

5.5.2. Remediation

Inspex suggests removing the `_withUpdate` variable in the `add()` and `set()` functions and always calling the `massUpdatePools()` function before updating `totalAllocPoint` variable as shown in the following example:

SamoyedMasterChef.sol

```
function add(
    uint256 _allocPoint,
    IBEP20 _lpToken,
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external onlyOwner {
    massUpdatePools();
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({
```

```
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accSmoyPerShare: 0,
                minDepositFeeRate: _minDepositFeeRate,
                maxDepositFeeRate: _maxDepositFeeRate
            })
        );
}
```

SamoyedMasterChef.sol

```
function set(
    uint256 _pid,
    uint256 _allocPoint,
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external onlyOwner {
    massUpdatePools();
    uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].minDepositFeeRate = _minDepositFeeRate;
    poolInfo[_pid].maxDepositFeeRate = _maxDepositFeeRate;
    if (prevAllocPoint != _allocPoint) {
        totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
    }
}
```

5.6. Centralized Control of State Variable

ID	IDX-006
Target	SamoyedMasterChef
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, these actions can only be performed by the contract owner.
Status	Resolved SamoyedFinance team has resolved this issue as suggested by implementing a timelock mechanism. The `SamoyedMasterChef` contract is now owned by the `Timelock` contract with 1 day minimum delay. `Timelock` contract: https://bscscan.com/address/0xe355bbb2ebc9986b16a42a8748c729ee849baf8 `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D02378670119453530478288AEe67b807e2a Ownership transfer of `SamoyedMasterChef` to `Timelock` contract: https://bscscan.com/tx/0xa6a2fb3238d5daf35e3bca5dabfb61665b83acd528381d793ac20cb0bfcb25f4#eventlog Platform users should monitor the execution of functions in the timelock and act accordingly.

5.6.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
SamoyedMasterChef.sol (L:100)	SamoyedMasterChef	updateMultiplier()	onlyOwner
SamoyedMasterChef.sol (L:110)	SamoyedMasterChef	add()	onlyOwner
SamoyedMasterChef.sol (L:135)	SamoyedMasterChef	set()	onlyOwner
SamoyedMasterChef.sol (L:328)	SamoyedMasterChef	updateMinimumSmoy()	onlyOwner
SamoyedMasterChef.sol (L:333)	SamoyedMasterChef	updateSmoyPerBlock()	onlyOwner
Ownable.sol (L:53)	SamoyedMasterChef	renounceOwnership()	onlyOwner
Ownable.sol (L:61)	SamoyedMasterChef	transferOwnership()	onlyOwner

Please note that the `Ownable` contract is inherited from the OpenZeppelin's library.

5.6.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a sufficient amount of time, e.g., 24 hours

5.7. Design Flaw in massUpdatePools() Function

ID	IDX-007
Target	SamoyedMasterChef
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePools()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `massUpdatePools()` is unusable.
Status	Resolved * SamoyedFinance team has resolved this issue in the deployed contract on mainnet by adding an `enabled` parameter to the pool to check whether the pool has ended or not, and adding condition in the `massUpdatePools()` function to update only the pools that have not ended. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D02378670119453530478288AEe67b807e2a However, the fix implemented leads to another issue on reward miscalculation in `allocPoint` and `enabled` parameters. This is because even if the pool is disabled (`enabled=false`), the disabled pool still has effect on other pools since the pool reward portion (`allocPoint`) can be more than zero, leading to lower reward to other pools (`totalAllocPoint`). For the new issue, SamoyedFinance team has clarified that the team will set a pool with `enabled = false` only for the pool with `allocPoint = 0` to prevent this issue.

5.7.1. Description

The `massUpdatePools()` function executes the `updatePool()` function, which is a state modifying function for all added pools as shown below:

SamoyedMasterChef.sol

```
function massUpdatePools() public nonReentrant {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        updatePool(pid);
    }
}
```

With the current design, the added pools cannot be removed. They can only be disabled by setting the `pool.allocPoint` to 0. Even if a pool is disabled, the `updatePool()` function for this pool is still called. Therefore, if new pools continue to be added to this contract, the `poolInfo.length` will continue to grow and this function will eventually be unusable due to excessive gas usage.

5.7.2. Remediation

Inspex suggests making the contract capable of removing unnecessary or ended pools to reduce the loop round in the `massUpdatePools()` function.

5.8. Unchecked Deposit Fee Value

ID	IDX-008
Target	SamoyedMasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium The owner of SamoyedMasterChef can set `minDepositFeeRate` and `maxDepositFeeRate` to an inappropriate amount, causing the `deposit()` function unusable when the fee amount is more than 100%. **Likelihood:** Low It is very unlikely that the contract owner will set the deposit fee amount to an improper value.
Status	Resolved SamoyedFinance team has resolved this issue as suggested in the deployed contract on mainnet. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D0237867011945353O478288AEe67b807e2a

5.8.1. Description

In the `SamoyedMasterChef` contract, `add()` and `set()` functions can be used to add a new farming pool and update the pool's parameters. The `minDepositFeeRate` and the `maxDepositFeeRate` parameters are used as fee rates that the user will be charged when making a deposit.

SamoyedMasterChef.sol

```
function add(
    uint256 _allocPoint,
    IBEP20 _lpToken,
    bool _withUpdate,
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external 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,
            accSmoyPerShare: 0,
            minDepositFeeRate: _minDepositFeeRate,
            maxDepositFeeRate: _maxDepositFeeRate
        })
    );
}

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

When users deposit `lpToken` to the contract, the deposited amount will be deducted by `collectDepositFee()` function.

SamoyedMasterChef.sol

```
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
    require(_pid != 0, "SamoyedMasterChef: deposit SMOY by staking");

    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
        uint256 pending = user.amount.mul(pool.accSmoyPerShare).div(1e12).sub(user.rewardDebt);
        if (pending > 0) {
```

```
            safeSmoyTransfer(msg.sender, pending);
        }
    }
    if (_amount > 0) {
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        uint256 amountAfterFee = collectDepositFee(_pid, _amount);
        user.amount = user.amount.add(amountAfterFee);
    }
    user.rewardDebt = user.amount.mul(pool.accSmoyPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

The `collectDepositFee()` uses the `_amount` multiplied with the `depositFeeRate` to calculate the fee. When the `depositFeeRate` is more than 100%, the `deposit()` function will be unusable.

SamoyedMasterChef.sol

```
function collectDepositFee(uint256 _pid, uint256 _amount) private returns (uint256 amount) {
    PoolInfo storage pool = poolInfo[_pid];
    if (pool.minDepositFeeRate > 0 || pool.maxDepositFeeRate > 0) {
        uint256 userBalance = smoy.balanceOf(msg.sender);
        uint256 depositFeeRate = userBalance < minimumSmoy ? pool.maxDepositFeeRate : pool.minDepositFeeRate;

        uint256 fee = _amount.mul(depositFeeRate).div(10000);
        pool.lpToken.transfer(feeCollectorAddr, fee);
        return _amount.sub(fee);
    }

    return _amount;
}
```

5.8.2. Remediation

Inspex suggests limiting the `minDepositFeeRate` and the `maxDepositFeeRate` in the `add()` and the `set()` as shown in the following example:

SamoyedMasterChef.sol

```
uint256 public LIMIT_MIN_DEPOSIT_FEE_RATE = 10000; // limit min fee 100%
uint256 public LIMIT_MAX_DEPOSIT_FEE_RATE = 10000; // limit max fee 100%

function add(
    uint256 _allocPoint,
    IBEP20 _lpToken,
    bool _withUpdate,
```

```
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external onlyOwner {
    require(_minDepositFeeRate <= LIMIT_MIN_DEPOSIT_FEE_RATE, "Fee rate is too high");
    require(_maxDepositFeeRate <= LIMIT_MAX_DEPOSIT_FEE_RATE, "Fee rate is too high");

    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accSmoyPerShare: 0,
            minDepositFeeRate: _minDepositFeeRate,
            maxDepositFeeRate: _maxDepositFeeRate
        })
    );
}

// Update the given pool's SMOY allocation point. Can only be called by the owner.
function set(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate,
    uint256 _minDepositFeeRate,
    uint256 _maxDepositFeeRate
) external onlyOwner {
    require(_minDepositFeeRate <= LIMIT_MIN_DEPOSIT_FEE_RATE, "Fee rate is too high");
    require(_maxDepositFeeRate <= LIMIT_MAX_DEPOSIT_FEE_RATE, "Fee rate is too high");

    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].minDepositFeeRate = _minDepositFeeRate;
    poolInfo[_pid].maxDepositFeeRate = _maxDepositFeeRate;
```

```
    if (prevAllocPoint != _allocPoint) {
        totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
    }
}
```

5.9. Insufficient Logging for Privileged Functions

ID	IDX-009
Target	SamoyedMasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	**Resolved** SamoyedFinance team has resolved this issue as suggested in the deployed contract on mainnet. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D02378670119453530478288AEe67b807e2a

5.9.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts to the platform.

For example, the owner can set the amount of $SMOY per block by executing `updateSmoyPerBlock()` function in the `SamoyedMasterChef` contract, and no event is emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function
SamoyedMasterChef.sol (L:100)	SamoyedMasterChef	updateMultiplier()
SamoyedMasterChef.sol (L:110)	SamoyedMasterChef	add()
SamoyedMasterChef.sol (L:135)	SamoyedMasterChef	set()
SamoyedMasterChef.sol (L:317)	SamoyedMasterChef	dev()
SamoyedMasterChef.sol (L:323)	SamoyedMasterChef	updateFeeCollector()
SamoyedMasterChef.sol (L:328)	SamoyedMasterChef	updateMinimumSmoy()

SamoyedMasterChef.sol (L:333)	SamoyedMasterChef	updateSmoyPerBlock()

5.9.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

SamoyedMasterChef.sol

```
333 event SmoyPerBlock(uint 256);
334 function updateSmoyPerBlock(uint256 _smoyPerBlock) external onlyOwner {
335     smoyPerBlock = _smoyPerBlock;
336     emit SmoyPerBlock(smoyPerBlock);
337 }
```

5.10. Unsupported Design for Deflationary Token

ID	IDX-010
Target	SamoyedMasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SamoyedFinance team has resolved this issue as suggested in the deployed contract on mainnet. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D02378670119453530478288AEe67b807e2a

5.10.1. Description

In the `SamoyedMasterChef` contract, the users can deposit their tokens to acquire rewards ($SMOY). The deposited tokens can be a normal token or LP token depending on the pools added by the contract owner.

However, in the `deposit()` function, an issue could arise when the pool uses a deflationary token (the token that reduces the circulating supply itself when it is transferred).

This means the `_amount` that the user deposit will be reduced due to the deflationary mechanism, but the contract recognizes it as the full amount as in line 216.

SamoyedMasterChef.sol

```
201 function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
202     require(_pid != 0, "SamoyedMasterChef: deposit SMOY by staking");
203
204     PoolInfo storage pool = poolInfo[_pid];
205     UserInfo storage user = userInfo[_pid][msg.sender];
206     updatePool(_pid);
207     if (user.amount > 0) {
208         uint256 pending =
    user.amount.mul(pool.accSmoyPerShare).div(1e12).sub(user.rewardDebt);
209         if (pending > 0) {
210             safeSmoyTransfer(msg.sender, pending);
211         }
212     }
```

```
    if (_amount > 0) {
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        uint256 amountAfterFee = collectDepositFee(_pid, _amount);
        user.amount = user.amount.add(amountAfterFee);
    }
    user.rewardDebt = user.amount.mul(pool.accSmoyPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

The failure of recognizing the token amount could lead to the following scenarios:

Scenario 1: Unable to withdraw staking tokens

Assuming that there is a pool in the `SamoyedMasterChef` contract which receives a deflationary token ($TOKEN) with 10% burn rate when the token is transferred.

Currently, there is only User A who stakes $TOKEN to the $TOKEN pool in the `SamoyedMasterChef` contract.

Holder	Balance
User A	100

Total $TOKEN in the `SamoyedMasterChef` contract: 90

User B deposits 100 $TOKEN to the $TOKEN pool in the `SamoyedMasterChef` contract. The `SamoyedMasterChef` contract will receive 90 $TOKEN since $TOKEN is 10% deduction from the deflationary mechanism, in this case 10 $TOKEN.

Holder	Balance
User A	100
User B	100

Total $TOKEN in the `SamoyedMasterChef` contract: 180

User B then withdraws 100 $TOKEN from the `SamoyedMasterChef` contract. The `SamoyedMasterChef` contract will validate whether the withdrawn `_amount` exceeds the `user.amount` or not.

SamoyedMasterChef.sol

```
function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
    require(_pid != 0, "SamoyedMasterChef: withdraw SMOY by unstaking");
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    require(user.amount >= _amount, "SamoyedMasterChef: withdraw: not good");
```

```
    updatePool(_pid);
    uint256 pending =
user.amount.mul(pool.accSmoyPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        safeSmoyTransfer(msg.sender, pending);
    }
    if (_amount > 0) {
        user.amount = user.amount.sub(_amount);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
    }
    user.rewardDebt = user.amount.mul(pool.accSmoyPerShare).div(1e12);
    emit Withdraw(msg.sender, _pid, _amount);
}
```

Since User B deposited 100 $TOKEN and the balance of $TOKEN in the contract is greater than 100, User B is allowed to withdraw 100 $TOKEN.

Holder	Balance
User A	100
User B	0

Total $TOKEN in the `SamoyedMasterChef` contract: 80

As a result, if User A decides to withdraw 100 $TOKEN, this transaction will be reverted since the balance in the contract is insufficient.

Scenario 2: Reward Calculation Exploit

Assuming that there is a pool in the `SamoyedMasterChef` contract which receives a deflationary token ($TOKEN) with 10% burn rate when the token is transferred.

Currently, there are several users who stake $TOKEN to the $TOKEN pool in the `SamoyedMasterChef` contract with a total supply of 100 $TOKEN.

User A deposits 100 $TOKEN to the contract, and the contract receives 90 $TOKEN due to the deflationary mechanism, resulting in a total supply of 190 $TOKEN.

After that, User A withdraws 100 $TOKEN from staking, the `SamoyedMasterChef` contract will then calculate the rewards as in line 208.

SamoyedMasterChef.sol

```
201 function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
202     require(_pid != 0, "SamoyedMasterChef: deposit SMOY by staking");
```

```
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
        uint256 pending =
user.amount.mul(pool.accSmoyPerShare).div(1e12).sub(user.rewardDebt);
        if (pending > 0) {
            safeSmoyTransfer(msg.sender, pending);
        }
    }
    if (_amount > 0) {
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this),
_amount);
        uint256 amountAfterFee = collectDepositFee(_pid, _amount);
        user.amount = user.amount.add(amountAfterFee);
    }
    user.rewardDebt = user.amount.mul(pool.accSmoyPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

During the calculation, the reward is affected by the total amount of $TOKEN (`lpSupply`) as in line 187.

SamoyedMasterChef.sol

```
182 function updatePool(uint256 _pid) public {
183     PoolInfo storage pool = poolInfo[_pid];
184     if (block.number <= pool.lastRewardBlock) {
185         return;
186     }
187     uint256 lpSupply = pool.lpToken.balanceOf(address(this));
188     if (lpSupply == 0) {
189         pool.lastRewardBlock = block.number;
190         return;
191     }
192     uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
193     uint256 smoyReward =
multiplier.mul(smoyPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
194     smoy.mintTo(devAddr, smoyReward.div(10));
195     smoy.mintTo(address(kennel), smoyReward);
196     pool.accSmoyPerShare =
pool.accSmoyPerShare.add(smoyReward.mul(1e12).div(lpSupply));
197     pool.lastRewardBlock = block.number;
198 }
```

Since the `SamoyedMasterChef` contract registers the `user.amount` of User A as 100 $TOKEN, the withdrawn $TOKEN amount will be 100, resulting in reducing the total amount of $TOKEN in the contract to 90 $TOKEN.

Hence, the value of `pool.accSmoyPerShare` can be increased dramatically by manipulating the total amount of $TOKEN (`lpSupply`) to be as low as possible.

User A can repeatedly execute `withdraw()` and `deposit()` functions to drain the $TOKEN in the contract until it is as low as possible, for example, 1 `wei`, causing `accSmoyPerShare` state to be overly inflated, so the users can claim an exceedingly large amount of reward ($SMOY) from the contract.

However, since only LP tokens are planned to be used in `SamoyedMasterChef` pools, there is no direct impact for this issue.

5.10.2. Remediation

Inspex suggests modifying the logic of the `deposit()` function to validate the amount of the received token from the user instead of using the value of `_amount` parameter directly.

SamoyedMasterChef.sol

```
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
    require(_pid != 0, "SamoyedMasterChef: deposit SMOY by staking");

    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
        uint256 pending =
user.amount.mul(pool.accSmoyPerShare).div(1e12).sub(user.rewardDebt);
        if (pending > 0) {
            safeSmoyTransfer(msg.sender, pending);
        }
    }
    if (_amount > 0) {
        uint256 currentBal = pool.lpToken.balanceOf(address(this));
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this),
_amount);
        uint256 receivedAmount = pool.lpToken.balanceOf(address(this)) -
currentBal;
        uint256 amountAfterFee = collectDepositFee(_pid, receivedAmount);
        user.amount = user.amount.add(amountAfterFee);
    }
    user.rewardDebt = user.amount.mul(pool.accSmoyPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

5.11. Improper Function Visibility

ID	IDX-011
Target	SamoyToken KennelClub SamoyedMasterChef
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact SamoyedFinance team has acknowledged this issue. The team however has resolved this issue partially since `SamoyedMasterChef` and `KennelClub` contract are fixed, but `SamoyToken` contract is not. `SamoyedMasterChef` contract: https://bscscan.com/address/0x5D21D023786701194535304782884Ee67b807e2a `KennelClub` Contract: https://bscscan.com/address/0x1364e039de60522aef045095823148e5e20f649a

5.11.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

For example, the following source code shows that the `mint()` function of the `SamoyToken` contract is set to public and it is never called from any internal function.

SamoyToken.sol

```
function mint(uint256 amount) public onlyOwner returns (bool) {
    _mint(_msgSender(), amount);
    return true;
}
```

Public

The following table contains all functions that have `public` visibility and are never called from any internal function.

File	Contract	Function
SamoyToken.sol (L:146)	SamoyToken	increaseAllowance()
SamoyToken.sol (L:165)	SamoyToken	decreaseAllowance()
SamoyToken.sol (L:182)	SamoyToken	mint()
SamoyToken.sol (L:195)	SamoyToken	burn()
KennelClub.sol (L:13)	KennelClub	mint()
KennelClub.sol (L:18)	KennelClub	burn()
KennelClub.sol (L:18)	KennelClub	safeSmoyTransfer()
SamoyedMasterChef.sol (L:201)	SamoyedMasterChef	deposit()
SamoyedMasterChef.sol (L:223)	SamoyedMasterChef	withdraw()
SamoyedMasterChef.sol (L:243)	SamoyedMasterChef	enterStaking()
SamoyedMasterChef.sol (L:264)	SamoyedMasterChef	leaveStaking()
SamoyedMasterChef.sol (L:284)	SamoyedMasterChef	emergencyWithdraw()

5.11.2. Remediation

Inspex suggests changing all functions' visibility to `external` if they are not called from any internal function as shown in the following example:

SamoyToken.sol

```
function mint(uint256 amount) external onlyOwner returns (bool) {
    _mint(_msgSender(), amount);
    return true;
}
```

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspex

CYBERSECURITY PROFESSIONAL SERVICE