STAYSAFU
AUDIT

November 4TH, 2022

TokenBurnable
Pools & ERC20 Tokens

TABLE OF CONTENTS

address(0)

B. [FIXED]STT-1: startTime can be adjusted unlimitedly

C. [FIXED]UNA-1: Unaccounted branch from lastRewardTime setter

D. [FIXED]LREW-1: Unneeded logic in lastRewardTime sub-branch

E. [FIXED]LREW-2: Ability to set lastRewardTime to uncapped future date

F. [FIXED]ISST-1: isStarted bool always resolves to tru

G. BLCK-2: Using an input for block.timestamp

H. [FIXED]POOL-1: Pool total rewards update miscalculating

I. INIT-2: Initializing msg.sender as memory variable

J. [FIXED]DIST-1: Insufficient reward distribution possibility

K. [FIXED]LOOP-1: For loop initialized value that is default

AUDIT SUMMARY

This report was written for ToeknBurnable in order to find flaws and vulnerabilities in the ToeknBurnable project's source code, as well as any contract dependencies that weren't part of an officially recognized library.

A comprehensive examination has been performed, utilizing Static Analysis, Manual Review, and ToeknBurnable Deployment techniques. The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors
- Assessing the codebase to ensure compliance with current best practices and industry standards
- Ensuring contract logic meets the specifications and intentions of the client
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders
- Through line-by-line manual review of the entire codebase by industry expert

AUDIT OVERVIEW

PROJECT SUMMARY

Project name	ToeknBurnable
Description	Numbers betting game and a clone of tombfork (bomb.money) farm, stake and bond.
Platform	TBD
Language	Solidity
Codebase	Pools and supporting ERC20 tokens

FINDINGS SUMMARY

Vulnerability	Total
Critical	1
Major	4
Medium	6
Minor	9
Informational	0

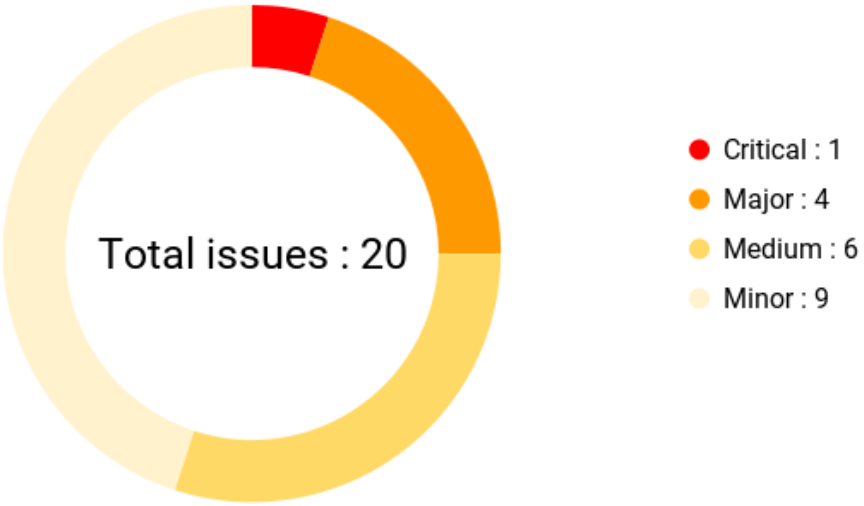
EXECUTIVE SUMMARY

ToeknBurnable is building a series of incentivized pools, where users can deposit tokens in exchange for rewards. The pools are initialized by the operator and freely interacted with by users to deposit, withdraw, and claim rewards through these actions. Each individual pool is limited to a single token and the reward distribution is based on a combination of per second reward rate and the proportion of total reward allocation assigned to each pool.

The share and token contracts follow the ERC20 standard with additional logic such as querying prices and distributing rewards.

There have been a series of major issues related to the codebase highlighted through the audit. The major issues that were found include:

- Possible initialisation of zero addresses without setter to change state variables.
- Double counting risks if pools are activated in a loop in the reward calculation logic.
- Insufficient underlying token balance leading to rewards being distributed to users incorrectly.
- No call limits on reward distribution of ERC20 token, which could lead to key stakeholders being remunerated more than once.



AUDIT FINDINGS

Code	Title	Severity
BOOL-1	Returning bool always true	Minor
DREW-1	No call limit to distributeReward	Major
SUP-1	totalSupply cap on token	Minor
SUP-2	totalSupply cap on shares	Minor
AIR-1	Airdrop funds can be sent to address(0)	Medium

TRE-1	treasuryFund can be set to address(0)	Medium
BLCK-1	Creating variable for block.timestamp	Minor
REW-1	Address(0) removable in claimRewards	Minor
INIT-1	Token can be initialized as address(0)	Critical
STT-1	startTime can be adjusted unlimitedly	Minor
UNA-1	Unaccounted branch from lastRewardTime setter	Medium
LREW-1	Unneeded logic in lastRewardTime sub-branch	Medium
LREW-2	Ability to set lastRewardTime to uncapped future date	Medium
ISST-1	isStarted bool always resolves to true	Medium
BLCK-2	Using an input for block.timestamp	Minor
POOL-1	Pool total rewards update miscalculating	Major
INIT-2	Initializing msg.sender as memory variable	Minor

DIST-1	Insufficient reward distribution possibility	Major
LOOP-1	For loop initialized value that is default	Minor
OPE-1	Operator can be set to address(0)	Major

[FIXED] FINDINGS for Tokens

[FIXED] BOOL-1 | Returning bool always true

Description

The mint function returns a bool if the balanceAfter is more than before, which should always be the case if the ERC20 implementation being used follows standard practice. The logic to get the balances and then return the bool is not required as a successful _mint call would always result in the balance being more than before.

Recommendation

Remove the logic to get balances and return true if desired:

```
_mint(recipient_, amount_);
return true;
```

[FIXED] DREW-1 | No call limit to distributeReward

Description

The distribute reward function does not check if rewardPoolDistributed is true, which means it can be called multiple times to distribute the defined reward pools to each party. This could drastically inflate the supply of tokens and there should be a check in place to prevent it.

Recommendation

Add bool check when the function is called to ensure rewardPoolDistributed is false.

```
require(!rewardPoolDistributed, “REWARDS_SENT”);
```

[FIXED] SUP-1 | totalSupply cap on token contract

Description

There is no cap on the total supply being imposed by the mint function, which may be the desired functionality. However, if there is a desire to add a supply cap then the logic will need to be added to as a state variable and to the mint function as a check before minting.

Recommendation

Add MAX_SUPPLY constant to state variables.

```
Uint256 public constant MAX_SUPPLY = <number>;
```

Add total supply check to the mint function

```
require(totalSupply() + amount_ < MAX_SUPPLY, “MAX_REACHED”);
```

[FIXED] SUP-2 | totalSupply cap on shares

Description

There is no cap on the total supply being imposed by the mint function, which may be the desired functionality. However, if there is a desire to add a supply cap then the logic will need to be added to as a state variable and to the mint function as a check before minting.

Recommendation

Add MAX_SUPPLY constant to state variables:

```
Uint256 public constant MAX_SUPPLY = <number>;
```

Add total supply check to the mint function:

```
require(totalSupply() + amount_ < MAX_SUPPLY, “MAX_REACHED”);
```

[FIXED] AIR-1 | Airdrop funds can be sent to address(0)

Description

There are checks that the office and genesis pool are not zero addresses in the distributeReward() function but there are no checks for airdrop. This will mean that the airdrop tokens could be sent to the zero address and would be unrecoverable.

Recommendation

Add address(0) check to distributeReward() function:

```
require(_air_drop != address(0), “!_airdrop”);
```

[FIXED] TRE-1 | treasuryFund can be set to address(0)

Description

There are no checks to ensure the new address being used for the treasuryFund in setTreasuryFund() is not a zero address. This means, the treasuryFund could be set incorrectly via the setter, which is preventable.

Recommendation

Add check that the communityFund input is not a zero address

```
require(_communityFund != address(0), “!_communityFund”);
```

BLCK-1 | Creating variable for block.timestamp

Description

In the unclaimedTreasuryFund() and unclaimedDevFund() functions the _now memory variable is initialized and used to store block.timestamp. This uses extra gas and is not required as block.timestamp can be used in its place throughout the function.

Recommendation

Use block.timestamp for all inputs and remove the initialization of _now

```
require(_communityFund != address(0), “!_communityFund”);
```

[FIXED] REW-1 | Address(0) removable in claimRewards

Description

The claimRewards() function checks if the communityFund and devFund are set to address(0) before acting. This logic could be removed if the address(0) checks are added to all setters for devFund and communityFund, as each state variable will be initialized correctly in the constructor and if they are changed in a setter the value could never be set to a zero address.

Recommendation

Remove the zero address checks (if address(0) require checks are added to the setters for community fund and dev fund)

```
if(pending > 0) {}
```

[FIXED] FINDINGS for Reward & Genesis Pools

The code being used for reward and genesis pools resemble each other, as the baseline functionality is the same. In order to reduce repetition, the issues highlighted are prefaced with the comment that they can be found in both contracts. Each of these cases also have two recommendations referred to each contract.

[FIXED] INIT-1 | Token can be initialized as address(0)

Description

This issue is found in both contracts.

In the constructor in Rewards Pool, the LCS token state variable will be initialized as a zero address if the address being input is a zero address - as by default the lcs state variable is a zero address. This would make the safeLCSTransfer function redundant as the lcs.safeTransfer call will be to a zero address.

There are no address setters in the contract, meaning if the contract is deployed with a zero address then user rewards will be blocked safeLCSTransfer is unable to be used.

In the constructor in Genesis Pool, the LCK token state variable will be initialized as a zero address due to the same reasons as above. The safeLCKTransfer function will also experience the same error.

Recommendation

Add require check that _lcs is not zero address in Reward Pool

```
require(_lcs != address(0), “BAD_ADDR”);
```

Add require check that _lck is not zero address in Genesis Pool

```
require(_lck != address(0), “BAD_ADDR”);
```

[FIXED] STT-1 | startTime can be adjusted unlimitedly before startTime has passed

Description

This issue is found in both contracts.

The setStartTime() function in Reward Pool does not check if the startTime does not equal a zero value or if a bool such as startTimeSet. This means that the startTime can be changed an unlimited amount of times until the current time is more than its value.

This may be a feature that is desired by the team, to enable customisation to startTime if it has been set and timings changes. However, if it is not a desired feature then it should be edited.

In Genesis Pool, the startTime can be adjusted due to the same reasons as above.

Recommendation

Add require check if poolStartTime equals 0 in Reward Pool

```
require(poolStartTime == 0, “ALREADY_SET”);
```

Add require check if poolStartTime equals 0 in Genesis Pool

```
require(poolStartTime == 0, “ALREADY_SET”);
```

[FIXED] LREW-1 | Unneeded logic in lastRewardTime sub-branch

Description

This issue is found in both contracts.

In Reward Pool, add() is called and the lastRewardTime is set based on the relationship between block.timestamp and poolStartTime. The first if statement checks whether the timestamp is less than poolStartTime and provides two sub branches that both set lastRewardTime to poolStartTime if true.

These branches are: (1) if lastRewardTime is zero then set to poolStarttime, and (2) if lastRewardTime is less than poolStartTime then set to poolStartTime.

As this logic is repeated, we can remove the sub-branches and set the lastRewardTime to poolStartTime if block.timestamp is less than poolStartTime.

In Genesis Pool, the same repeated logic issue exists.

Recommendation

Remove the sub-branch logic from the first if statement in Reward Pool

```
if(block.timestamp < poolStartTime) {
    _lastRewardTime = poolStartTime;
}
```

Remove the sub-branch logic from the first if statement in Genesis Pool

```
if(block.timestamp < poolStartTime) {
    _lastRewardTime = poolStartTime;
}
```

[FIXED] UNA-1 | Unaccounted branch from lastRewardTime setter

Description

This issue is found in both contracts.

In the add function, the logic that sets the lastRewardTime does not account for a scenario. If the current time is less than poolStartTime and the lastRewardTime is more than poolStartTime then the value used for lastRewardTime will remain unchanged.

This means when the isStarted bool check occurs, isStarted will be set to true even though the poolStartTime is less than the block.timestamp. By condensing the logic, as suggested in X-4 we are able to prevent this from happening. As if the block.timestamp is less than poolStartTime, the lastRewardTime will always be set to poolStartTime.

In Genesis Pool, the same unaccounted branch exists.

Recommendation

Remove the sub-branch logic from the first if statement in Reward Pool as suggested above:

```
if(lastRewardTime < poolStartTime) {
    lastRewardTime = poolStartTime;
}
```

Remove the sub-branch logic from the first if statement in Genesis Pool as suggested above:

```
if(block.timestamp < poolStartTime) {
    _lastRewardTime = poolStartTime;
}
```

[FIXED] LREW-2 | Ability to set lastRewardTime to uncapped future date

Description

This issue is found in both contracts.

In the else branch of add(), where block.timestamp does not equal poolStartTime we set lastRewardTime. We check if lastRewardTime equals 0 or if it is less than block.timestamp and if either case is true then lastRewardTime is set to block.timestamp.

However, if lastRewardTime is more than block.timestamp then it will not be changed. This means it could be set to an unrestricted future time that is beyond the endTime for the raffle.

This may be purposeful design, however, it's unclear why it would be the case. If lastRewardTime is set to a time after the endTime then the logic for updates will not work as fromTime would always be more than block.timestamp within the timeframe up to poolEndTime.

In Genesis Pool, the same ability to bypass both trees of logic exists.

Recommendation

If block.timestamp is more than poolStartTime then set lastRewardTime to the block.timestamp in RewardPool

```
else {
        _lastRewardTime = block.timestamp
}
```

If block.timestamp is more than poolStartTime then set lastRewardTime to the block.timestamp in RewardPool

```
else {
        _lastRewardTime = block.timestamp
}
```

[FIXED] ISST-1 | isStarted bool always resolves to true

Description

This issue is found in both contracts.

When add() is called, the isStarted time bool is set based on whether the lastRewardTime is less or equal to poolStartTime or the lastRewardTime is less than block.timestamp.

The if statements before this setter either set lastRewardTime to block.timestamp or to poolStartTime meaning the check always results in true. This is only bypassed in the case highlighted above where the lastRewardTime can be set to an uncapped future date.

If the desired outcome is to always set isStarted to true then this should be done rather than using additional logic. However if isStarted should be set to true or false in different cases then the logic must be changed.

In Genesis Pool, isStarted also always results in being true.

Recommendation

Set bool to true without checks if desired or adjust the isStarted logic to result in different outcomes in Reward Pool.

Set bool to true without checks if desired or adjust the isStarted logic to result in different outcomes in Genesis Pool.

BLCK-2 | Using an input for block.timestamp

Description

This issue is found in both contracts.

The variable of _toTime is passed into getGeneratedRewards() function and in all of the cases where this function is called block.timestamp is used. This is understandable if the purpose is to make the code more readable, however, if using less gas is an important factor then using block.timestamp and removing the additional input variable from the function is suggested.

In the Genesis Pool, the same is true for getGeneratedReward() inputs.

Recommendation

Remove input from function parameters and change all instances _toTime is used to block.timestamp in Reward Pool:

```
Function getGeneratedReward(uint256 _fromTime) public view returns (uint256) {}
```

Remove input from function parameters and change all instances _toTime is used to block.timestamp in Genesis Pool:

```
Function getGeneratedReward(uint256 _fromTime) public view returns (uint256) {}
```

[FIXED] POOL-1 | Pool total rewards update miscalculating

Description

This issue is found in both contracts.

When updatePool() is called it will check if the pool has been started and if it has not the pool's allocPoint value will be added to totalAllocPoint. In the scenario where updatePool() is being called for the first time where block.timestamp is more than poolStartTime, the allocPoint total for the first pool being iterated on will be added to totalAllocPoint.

This will then repeat as each consecutive pool is iterated on and the totalAllocPoint value will increase. However, the calculation for rewards will always result in an incorrect value being returned.

For example: the first pool has an allocPoint of 1 and pool 2 has an allocPoint of 9. As we iterated on the first pool, the totalAllocPoint will change from 0 to 1 and the following if statement checking if totalAllocPoint > 0 will be true.

The _lcsReward calculation will take rewards per second up to the block.timestamp, multiply them by the pool's allocPoint, and divide the result by totalAllocPoint. At this stage, only pool's allocPoint has been added to totalAllocPoint meaning the full rewards for the period passed will be allocated to pool 1.

This error will repeatedly occur until the final pool's allocPoint value is added to totalAllocPoint. This means the wrong amount of rewards will be allocated to each pool each time and a double counting issue will occur. As the first pool receives all rewards for T1 and the second pool

may receive 9/10 of rewards for T1 (i.e. 1.9x rewards has been paid at this stage). This could be prevented if the totalAllocPoint was added to when pools are created in the add() function.

In the Genesis Pool, the same issue exits.

Recommendation

Add to the totalAllocPoint state variable when each pool is added in add(). Do this by removing the if (_isStarted) {} check Reward and Genesis Pool:

```
totalAllocPoint = totalAllocPoint.add(_allocPoint);
```

Remove the totalAllocPoint addition logic in the !pool.isStarted branch of the updatePool function in Reward and Genesis Pool:

```
if(!pool.isStarted) {
        pool.isStarted = true;
}
```

As block.timestamp will only be less than the lastRewardTime if we have not passed startTime or if an update has just happened, we can assume that the rest of the logic should be executed. Meaning, we remove the if statement and execute the reward logic in Reward and Genesis Pool.

INIT-2 | Initializing msg.sender as memory variable

Description

This issue is found in both contracts.

In the deposit() and withdraw() functions the msg.sender is set to a memory variable and used to pass into mappings and functions. This will make the function more costly in gas terms and the msg.sender can be used in place of the newly initialized memory variable name sender.

In Genesis Pool, the same use of a memory variable occurs.

Recommendation

Use msg.sender instead of the initialized variable sender in deposit() and withdraw() in Reward Pool.

Use msg.sender instead of the initialized variable sender in deposit() and withdraw() in Genesis Pool.

[FIXED] DIST-1 | Insufficient reward distribution possibility

Description

This issue is found in both contracts.

The lcsTransfer logic checks if the balance of lcs is more than zero and if the amount being requested is more than lcs balance, the balance is transferred instead. As this function is called to distribute pending rewards on deposits and withdrawals, the user may have 100 rewards pending but if the balance of lcs is 10 then this amount would be sent. As there are no checks, the user has been sent the incorrect amount but the contract believes the transaction was successful.

In Genesis Pool, the same issue occurs in safelckTransfer.

Recommendation

Add require statement that the _lcsBal is more than _amount and remove the checks in Reward Pool:

```
require(lcs.balanceOf(address(this)) > _amount, “NO_BALANCE”);
lcs.safeTransfer(_to, _amount)
```

Add require statement that the _lcsBal is more than _amount and remove the checks in Genesis Pool:

```
require(lck.balanceOf(address(this)) > _amount, “NO_BALANCE”);
lck.safeTransfer(_to, _amount)
```

[FIXED] LOOP-1 | For loop initialized value that is default

Description

This issue is found in both contracts.

The value for PID is initialized to equal 0 in a series of for loops.This logic is not required as the default value for a newly initialized uint is 0, which means it can be removed.

In Genesis Pool, the same issue occurs.

Recommendation

Remove the initialization of PID equals 0 in Reward and Genesis Pool:

```
for(uint256 pid; pid < length; ++pid) {}
```

[FIXED] OPE-1 | Operator can be set to address(0)

Description

In Genesis Pool, the setOperator function does not check if the _operator input is a zero address. This means that the operator could be set to a zero address and all functions that require operator status would no longer be accessible. To prevent this, a require statement is needed to check the input before the new value is set.

Recommendation

Add require check that the _operator input is not a zero add

```
require(_operator != address(0), “INVALID_ADDR”);
```

Global security warnings

These are safety issues for the whole project. They are not necessarily critical problems but they are inherent in the structure of the project itself. Potential attack vectors for these security problems should be monitored.

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement.

This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without StaySAFU's prior written consent.This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts StaySAFU to perform a security assessment.

This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project.
This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk.
StaySAFU's position is that each company and individual are responsible for their own due diligence and continuous security. StaySAFU's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or fun.

STAYSAFU AUDIT

November 4TH, 2022

TokenBurnable Manager & Raffle

TABLE OF CONTENTS

AUDIT SUMMARY

This report was written for ToeknBurnable in order to find flaws and vulnerabilities in the ToeknBurnable project's source code, as well as any contract dependencies that weren't part of an officially recognized library.

A comprehensive examination has been performed, utilizing Static Analysis, Manual Review, and ToeknBurnable Deployment techniques. The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors
- Assessing the codebase to ensure compliance with current best practices and industry standards
- Ensuring contract logic meets the specifications and intentions of the client
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders
- Through line-by-line manual review of the entire codebase by industry expert

AUDIT OVERVIEW

PROJECT SUMMARY

Project name	ToeknBurnable
Description	Numbers betting game and a clone of tombfork (bomb.money) farm, stake and bond.
Platform	TBD
Language	Solidity
Codebase	Raffle and Manager contracts

FINDINGS SUMMARY

Vulnerability	Total
Critical	0
Major	0
Medium	1
Minor	5
Informational	1

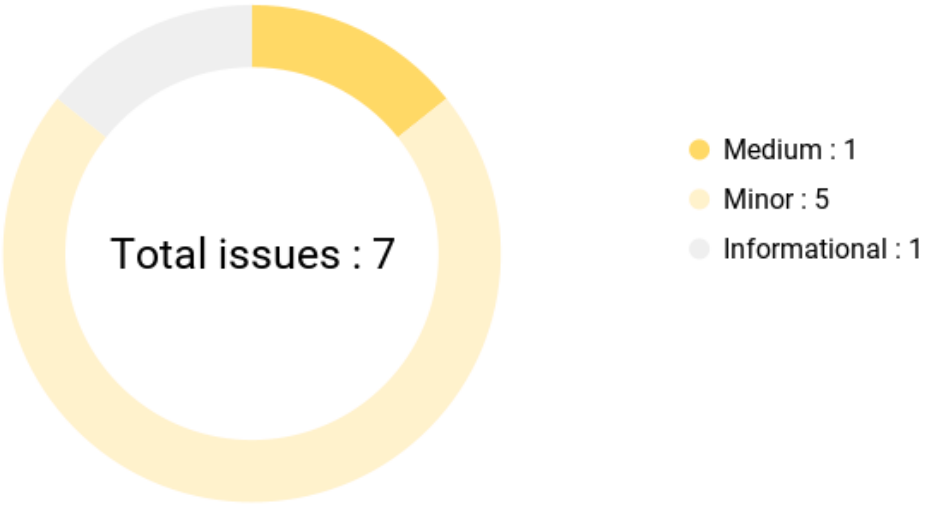
EXECUTIVE SUMMARY

ToeknBurnable Raffle & Manager are two interoperating smart contracts used to launch and manage a series of raffles. The Manager contract acts as the deployer, manager, and admin for all Raffle contracts deployed. Whilst the Raffle contract manages tickets purchased, winner outcomes, and reward distribution to the randomized winners.

The source of randomness used for each raffle is Chainlink, which is queried via the Manager contract, and the returned value is used to update Raffle. Raffle's are closed in the random number query process, and the value received is used to choose the winner by finding the link between the tokenID and address.

There have been no major or critical issues related to the codebase, and all findings listed here are medium or minor. The medium issue highlighted is a possible misuse of address based on the developer notes whilst the minor issues related to unused functions and, in some cases, require further clarification to confirm they need to be resolved.

AUDIT FINDINGS



Code	Title	Severity
GET-1	getPrice in IManager interface is a function that is not used	Minor
RAF-1	RaffleTotal is iterating on completed raffles rather than raffles created	Minor
SET-1	setState function unused and only callable by the deployer contract	Minor
DUP-1	Duplicate check in claimRewards	Minor
WRG-1	Possible wrong address being used for transfer	Medium

ITEM-1	Unused itemID logic for randomness	Info
ID-1	Id limit check imposed but multiple transactions are not prevented	Minor

GET-1 | getPrice in IManager interface is a function that is not used

Description

The IManager interface contains the logic for a getPrice function, however, it is not used in the logic of Raffle or Manager. If this function is not used throughout the contracts then it is possible to remove it.

Recommendation

Remove the getPrice logic from the interface

RAF-1 | RaffleTotal is iterating on completed raffles rather than raffles created

Description

The raffleTotal state variable is iterated when a raffle is completed, which may be the desired functionality. However, if the total is supposed to track raffles created rather than those that are successfully completed then it should be iterated on in createRaffle rather than resultRandom.

Recommendation

Move the iteration logic to createRaffle if the desire is to track created raffles rather than completed raffles.

SET-1 | setState function unused and only callable by the deployer contract

Description

The setState function in Raffle is only callable by the deployer contract, as it has a modifier restricting use to only the manager address. However, the Manager contract does not hold any logic that calls setState, which means the function can be removed.

Recommendation

If there is no intent to use setState then remove the setState function

DUP-1 | Duplicate check in claimRewards

Description

The msg.sender is compared with the winner in the claimRewards function to ensure the caller is eligible to receive rewards. However, the state of closed is also checked.

In the case that there is a winner, the winner state variable will change from address(0) to a valid address and closed will be true. In the case there is no winner, the winner variable will be a zero address as it's initialized to its default value and closed will be true.

Checking if the winner equals msg.sender should prevent exploits, as the winner will either be the correct address or address(0). Meaning the check of closed will always be true when the msg.sender is the winner and it will never be checked if the msg.sender is not the winner. In effect, the logic does not have any cases where it will revert and can be removed.

Recommendation

Remove the require check for the closed variable in claimReward().

WRG-1 | Possible wrong address being used for transfer

Description

The notes in the Raffle contract suggest that the transferFrom call in the tokenTransfer function is supposed to transfer the fee and the total amount to the office. If this is the case, address(this) is being used in the second transfer call and it should be changed to office instead.

Recommendation

Change address(this) to _office in tokenTransfer().

```
require(IERC20(token).transferFrom(msg.sender, _office, total.sub(fee)), “error”);
```

ITEM-1 | Unused itemID logic for randomness

Description

The winner is selected based on a random number received from chainlink. Whilst the user has the option to choose numbers for their raffle tickets (itemID) which I assume are inferred to be used in the selection process of the winner.

However, the random number received from Chainlink is actually used to select a winner from an NFT token ID that links to a user rather than a raffle ticket number (itemID).

In practice, the probability the user will have to win will remain the same regardless of the numbers they have chosen and from a UI perspective this would also not cause any issues. From an implementation perspective, buying 10 tickets as a user using a randomized list of numbers would lead to 10 consecutively numbered NFTs being minted rather than having a distributed number of tickets as expected from a raffle.

ID-1 | ID limit check imposed but multiple transactions are not prevented

Description

There is a limit that is imposed on the number of IDs being input on the mint function. However, if the limit that checks ids.length is less than 6 is supposed to be used to prevent the user from having more than 5 tickets in total for a raffle then it will need to be adjusted.

This is because a user can call the function multiple times in different blocks to accumulate more than 5 tickets. There is currently no totalTickets count being linked to the sender to restrict access otherwise.

Recommendation

If a ticket limit is desired then query the balanceOf the sender and require it is less than 6 plus the amount being minted

```
require(balanceOf(msg.sender) + ids.length < 6, “MAX_NUM”);
```

Global security warnings

These are safety issues for the whole project. They are not necessarily critical problems but they are inherent in the structure of the project itself. Potential attack vectors for these security problems should be monitored.

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement.

This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.
This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without StaySAFU's prior written consent.This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts StaySAFU to perform a security assessment.
This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance. This report should not be used in any way

to make decisions around investment or involvement with any particular project.

This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk.

StaySAFU's position is that each company and individual are responsible for their own due diligence and continuous security. StaySAFU's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or fun.