OBELISK

Part of Tibereum Group

AUDITING REPORT

Version Notes

Version	No. Pages	Date	Revised By	Notes
1.0	Total: 28	2021-06-04	Plemonade, Zapmore, Hebilicious	Final Audit

Audit Notes

Audit Date	2021-05-26 - 2021-06-02
Auditor/Auditors	Plemonade, Hebilicious
Auditor/Auditors Contact Information	tibereum-obelisk@protonmail.com
Notes	Specified code and contracts are audited for security flaws. UI/UX (website), logic, team, and tokenomics are not audited.
Audit Report Number	OB58856821

Disclaimer

This audit is not financial, investment, or any other kind of advice and is for informational purposes only. This report is not a substitute for doing your own research and due diligence. Obelisk is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Obelisk has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Obelisk is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. The audit is paid by the project but neither the auditors nor Obelisk has any other connection to the project and has no obligations other than to publish an objective report. Obelisk will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Obelisk assumes that the provided information and material were not altered, suppressed, or misleading. This report is published by Obelisk, and Obelisk has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Obelisk.

Obelisk Auditing

Defi is a relatively new concept but has seen exponential growth to a point where there is a multitude of new projects created every day. In a fast-paced world like this, there will also be an enormous amount of scams. The scams have become so elaborate that it's hard for the common investor to trust a project, even though it could be legit. We saw a need for creating high-quality audits at a fast phase to keep up with the constantly expanding market. With the Obelisk stamp of approval, a legitimate project can easily grow its user base exponentially in a world where trust means everything. Obelisk Auditing consists of a group of security experts that specialize in security and structural operations, with previous work experience from among other things, PricewaterhouseCoopers. All our audits will always be conducted by at least two independent auditors for maximum security and professionalism.

As a comprehensive security firm, Obelisk provides all kinds of audits and project assistance.

Table of Content

Project Information

Project Name	Southpark
Description	South Park — the only farm with unbelievably high APR and tokenomics smart enough to handle selling pressure on its native token — Kenny.
Website	https://southpark.finance/
Contact	@Kenny_defi
Contact information	@Kenny_defi on TG
Token Name(s)	South Park Token
Token Short	Kenny
Token Information	ERC20 Token that can be minted and burned, with taxed transfers and auto-liquidity feature.
Contract(s)	See Appendix A
CodeBase	https://gitlab.com/southparkfinance/southpark-contracts
Commit	1.d5b9903fe6550ee7d93d6888ef7cfc07c904bb81 2.8b8404b1f6b4281509ce2db1eebb2eb5e11971be
Code Language	Solidity
Chain	Polygon

Executive Summary

The audit of SouthPark was conducted by two of Obelisks' security experts between the 26th of May 2021 and the 2nd of June 2021. The contracts were audited before being published, and will then be compared to their deployed counterparts when they are deployed.

After finishing the full audit, Obelisk auditing can say that there were some security issues during the initial audit of the audited contracts from SouthPark. SouthPark project team either solved or commented on all issues found.

The team has not reviewed the UI/UX, logic, team, or tokenomics of the SouthPark project.

Please read the full document for a complete understanding of the audit.

Summary Table

Audited Part	Severity	Note
Minting Causes Unknown Growth Might Be Exploitable	Medium/Low Risk	Mitigated
Deposit Increases totalstake Wrong	High	Mitigated
Add reinvestReward Does Not Check If Received	Informational	See Comment
Transfer Always Taxes	Informational	Mitigated
Bypass lockup To An Extent	Medium/Low Risk	See Comment
Liquidity Centralization Risk	Low	See Comment
Swap Locking Variable	Informational	See Comment
Non-Withdrawable Matic	Low	See Comment
Third-Party Reliance	Informational	See Comment
Owner Bypass Liquidity Rules	Medium	Mitigated
Mintable And Burnable	High	Mitigated
Operator Staking Centralization	Informational	N/A

Introduction

Obelisk was commissioned by SouthPark on the 25th of May 2021 to conduct a comprehensive audit of SouthParks' farming project on Polygon. The following audit was conducted between the 26th of May 2021 and the 31st of May 2021 and delivered on the 2nd of June 2021. Two of Obelisk's security experts went through the related contracts using industry standards to find if any vulnerabilities could be exploited.

The comprehensive test was conducted in a specific test environment that utilized exact copies of the published contract. The auditors also conducted a manual visual inspection of the code to find security flaws that automatic tests would not find.

While conducting the audit, the Obelisk security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Obelisk assesses the risks and assigns a risk level to each section together with an explanatory comment. Take note that the comments from the project team are their opinion and not the opinion of Obelisk.

The initial audit found some high and medium severity issues. These issues were found both in the way deposits are calculated and the mintings of tokens. These issues were corrected by the SouthPark project team in order to make a safer project. The project team also commented with an explanation of issues that were not fixed so the reader can make their own opinion. There are also some informational findings that there for information only and wouldn't be any risk of losing funds.

Please see each section of the audit to get a full understanding of the audit.

Findings

Manual Analysis

Minting Causes Unknown Growth Might Be Exploitable

SEVERITY	Medium/Low
LOCATION	*MasterChef.sol -> 310-322*
RESOLVED	**Yes**

```
1      function rewardReferrers(uint baseAmount) internal {
2          address ref = msg.sender;
3          for (uint8 i = 0; i < referrerRewardRate.length; i++) {
4              ref = referrer[ref];
5              if (ref == address(0)) {
6                  break;
7              }
8
9              uint reward = baseAmount * referrerRewardRate[i] / 10000;
10             kenny.mint(ref, reward);
11             referrerReward[ref] += reward;
12         }
13     }
```

DESCRIPTION	Because the referral system mints depending on your baseAmount then having a big baseAmount(how much accumulated rewards you have from a pool when calling reinvest). Then this means that Kenny token supply will grow more when people have referrals at an unknown rate as transfers add to accumulated rewards on the Kenny pool. If an attacker saves up on Kenny they might be able to abuse that in conjunction with a flash loan(to make transfers so that more tokens will be minted), but one would probably have to sandwich attack the transfer and or have access to the operator account(able to turn off tax and gets 60% of transfers).
RECOMMENDATION	Have a supply that increases at a known rate or lock the extra mints for a certain amount of time to reduce the chance of such an attack vector.

MITIGATION	The project team has implemented a fix by not allowing the reinvest pool to mint out extra referral rewards. This makes the minting increase at a maximum known rate as it will just mint at a maximum small % extra to referrers. Note that this will happen somewhat sporadically as the minting happens on a claim/reinvest call.

Deposit Increases totalstake Wrong

SEVERITY	High
LOCATION	*MasterChef.sol -> 141-168*
RESOLVED	**Yes**

```
function deposit(uint pid, uint amount, address ref) external {
        require (pid != 0, "reinvest pool disallow deposit");

        PoolInfo storage pool = poolInfo[pid];
        UserInfo storage user = userInfo[pid][msg.sender];

        updatePool(pid);

        if (canClaim(pid)) {
            claim(pid);
        } else {
            lockup(pid);
        }

        if (amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), amount);

            if (pool.depositFeeBP > 0){
                uint depositFee = amount * pool.depositFeeBP / 10000;
                pool.lpToken.safeTransfer(feeAddress, depositFee);
                user.amount += amount - depositFee;
            } else {
                user.amount += amount;
            }
        }

        pool.totalStake += user.amount;
        user.rewardDebt = user.amount * pool.accKennyPerStake / 1e12;
```

DESCRIPTION	*pool.totalStake* uses the user.amount so if the user deposits 0(or any amount after first deposit) the pool.totalStake will increase more than it is supposed to leading to fewer rewards for everyone. With many users making multiple deposits this will be very noticeable as the rewards will go to 0 as pool.totalStake increases because of multi deposits or rewards claims.
RECOMMENDATION	Consider relying on balanceof inside the updatepool to be sure the pool amount actually has the same amount inside it or be very careful that the pool.totalStake is the right amount.
MITIGATED/COMMENT	The project team has fixed the issue by decrementing pool.totalstake with oldamount and then adding

	user.amount which fix the issue at hand.

Add reinvestReward Does Not Check If Received

SEVERITY	Informational
LOCATION	*MasterChef.sol -> 267-275*
RESOLVED	**No**

```
1    function addReinvestReward(uint amount) external override {
2        require(msg.sender == address(kenny), "KENNY only allowed");
3
4        PoolInfo storage pool = poolInfo[0];
5
6        if (pool.totalStake > 0) {
7            pool.accKennyPerStake += amount * 1e12 / pool.totalStake;
8        }
9    }
```

DESCRIPTION	The function assumes that Kenny is sent to the contract. This function thereby relies on the external source to transfer the amount. If the amount were to differ an attacker would be able to steal other users' Kenny. Also if the pool has 0 coins staked Kenny will accumulate without anyone being able to claim them in the pool.
RECOMMENDATION	Consider having the function do the transfer itself while checking it has received the correct amount to limit the complexity of cross-contract problems.
MITIGATION	The project team has acknowledged the issue and due to the fact that the contract is already deployed and non-upgradable, the project team determined the code to be left unchanged.

Transfer Always Taxes

SEVERITY	Informational
LOCATION	SouthParkToken.sol -> 67-89
RESOLVED	**Yes**

```
1
2  function _transfer(address sender, address recipient, uint256 amount) internal virtual override {
3    if (autoLiqEnabled && sender != owner() && sender != kennyEthPair) {
4      autoLiq();
5    }
6
7    if (transferTaxRate > 0 && kennyEthPair != address(0)) {
8      uint taxAmount = amount * transferTaxRate / 10000;
9      uint rewardAmount = taxAmount * taxRewardPartRate / 100;
10     uint autoLiqAmount = taxAmount - rewardAmount;
11
12     // transfer for auto liq
13     super._transfer(sender, address(this), autoLiqAmount);
14
15     // transfer reward to masterchef
16     _approve(address(this), owner(), rewardAmount);
17     super._transfer(sender, owner(), rewardAmount);
18     IHaveReinvestPool(owner()).addReinvestReward(rewardAmount);
19
20     amount -= taxAmount;
21   }
22
23   super._transfer(sender, recipient, amount);
24 }
```

DESCRIPTION	The token taxes (to reinvest pool, sent to add liquidity later) every address that is conducted externally. It taxes everyone including the *owner*(should correspond to MasterChef contract) and *kennyEthPair* but does not exchange for lp tokens before any other user/contract transfer. Lp tokens are then sent to the *Operator*. This means that every transfer of Kenny will incur a fee when using the Kenny token(ex buy, sell, deposit, withdraw)
RECOMMENDATION	Consider if this is expected behavior to tax for doing these actions as it might incur a big fee for users even when just buying and directly staking.
MITIGATED/COMMENT	The project team has acknowledged this is expected behavior.

Bypass Lockup To An Extent

SEVERITY	Medium/Low
LOCATION	MasterChef.sol ->141-193
RESOLVED	**No**

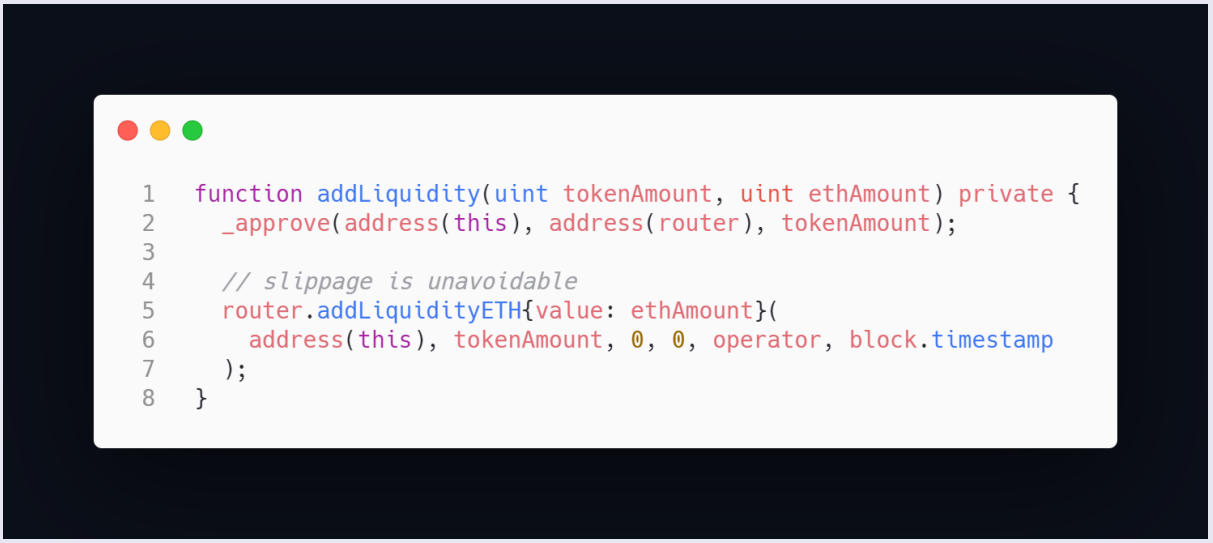
```
function deposit(uint pid, uint amount, address ref) external {
        require (pid != 0, "reinvest pool disallow deposit");

        PoolInfo storage pool = poolInfo[pid];
        UserInfo storage user = userInfo[pid][msg.sender];

        updatePool(pid);

        if (canClaim(pid)) {
            claim(pid);
        } else {
            lockup(pid);
        }

        if (amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), amount);

            if (pool.depositFeeBP > 0){
                uint depositFee = amount * pool.depositFeeBP / 10000;
                pool.lpToken.safeTransfer(feeAddress, depositFee);
                user.amount += amount - depositFee;
            } else {
                user.amount += amount;
            }
        }

        pool.totalStake += user.amount;
        user.rewardDebt = user.amount * pool.accKennyPerStake / 1e12;

        if (ref != address(0) && ref != msg.sender && referrer[msg.sender] == address(0)) {
            referrer[msg.sender] = ref;

            // level -1
            referrals[ref][0] += 1;
            referrals[ref][1] += referrals[msg.sender][0];
            referrals[ref][2] += referrals[msg.sender][1];

            // level -2
            address ref1 = referrer[ref];
            if (ref1 != address(0)) {
                referrals[ref1][1] += 1;
                referrals[ref1][2] += referrals[msg.sender][0];

                // level -3
                address ref2 = referrer[ref1];
                if (ref2 != address(0)) {
                    referrals[ref2][2] += 1;
                }
            }
        }

        emit Deposit(msg.sender, pid, amount);
    }
```

DESCRIPTION	1. By pre-depositing a small number of funds a user can make it so they only have to spend about 1

	block in the pool for rewards. For example, user deposits a second transaction just before the claim function can run and set a new 4-hour lock. (the claim function is what sets the lastclaimedblock) 2. Also, be careful with block.timestamp as it can be influenced in a short period of time. It's not a problem in the current implementation where hours are used.
RECOMMENDATION	Consider reworking the mechanism of locking deposit rewards for a certain period of time as users can just switch between wallets to get their rewards after 1 block.
MITIGATED/COMMENT	The project team has acknowledged the issue and considers it acceptable for people to use this if they want to.

Liquidity Centralization Risk

SEVERITY	Low
LOCATION	SouthParkToken.sol#134
RESOLVED	**No**



```
function addLiquidity(uint tokenAmount, uint ethAmount) private {
  _approve(address(this), address(router), tokenAmount);

  // slippage is unavoidable
  router.addLiquidityETH{value: ethAmount}(
    address(this), tokenAmount, 0, 0, operator, block.timestamp
  );
}
```

DESCRIPTION	The liquidity gets added to the operator, which eventually ends up with the operator controlling most of the token liquidity.
RECOMMENDATION	Add the liquidity to the contract balance, or to a specific contract designed to handle the liquidity. Alternatively use a long enough Timelock.
MITIGATED/COMMENT	The Project team acknowledged that the operator is the owner of those tokens and stated that it corresponds to their tokenomics as no liquidity locking was promised

Swap Locking Variable

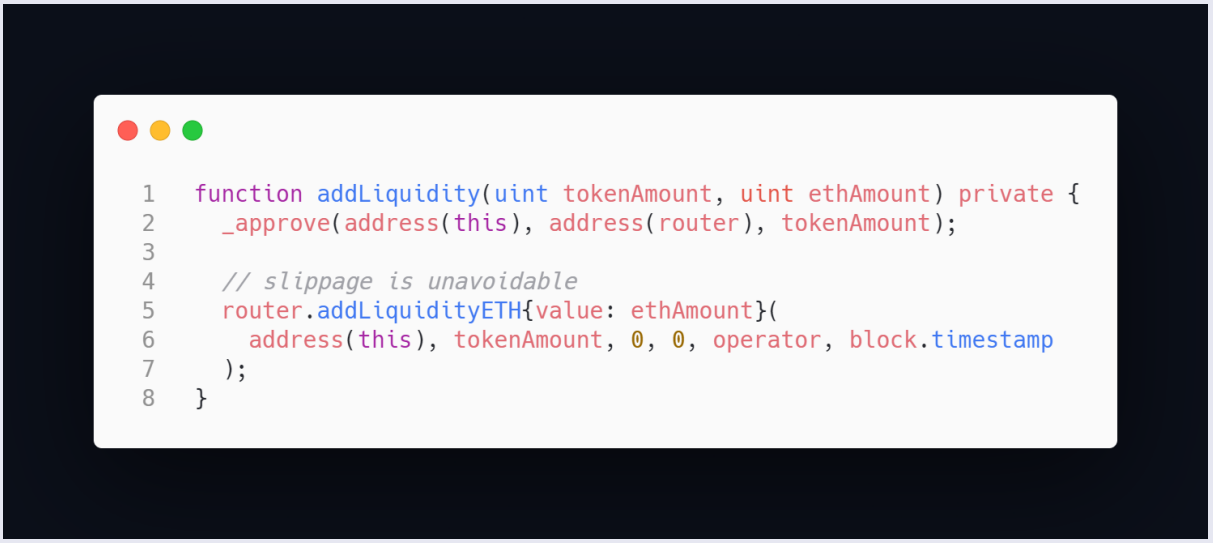
SEVERITY	Informational
LOCATION	SouthParkToken.sol 95-112
RESOLVED	**No**

```
function autoLiq() private {
    uint thisBalance = balanceOf(address(this));

    if (thisBalance >= autoLiqMinAmount) {
      autoLiqEnabled = false;

      uint liqAmount = autoLiqMinAmount;

      // split amount
      uint swapAmount = liqAmount / 2;
      uint tokenAmount = liqAmount - swapAmount;

      // save current eth balance
      uint ethBalance = address(this).balance;
      // make swap
      swapTokensForEth(swapAmount);
      uint ethAmount = address(this).balance - ethBalance;

      addLiquidity(tokenAmount, ethAmount);
      emit AutoLiq(tokenAmount, ethAmount);

      autoLiqEnabled = true;
    }
  }
```

DESCRIPTION	The swap locking variable can be modified by the operator and thereby has a bit of centralization risk.
RECOMMENDATION	Use a different variable that can’t be modified by the operator.
MITIGATED/COMMENT	The project team has acknowledged the issue and due to the fact that the contract is already deployed and non-upgradable, the project team determined the code to be left unchanged

Non-Withdrawable Matic

SEVERITY	Low
LOCATION	SouthParkToken.sol l133
RESOLVED	**No**



```
function addLiquidity(uint tokenAmount, uint ethAmount) private {
  _approve(address(this), address(router), tokenAmount);

  // slippage is unavoidable
  router.addLiquidityETH{value: ethAmount}(
    address(this), tokenAmount, 0, 0, operator, block.timestamp
  );
}
```

DESCRIPTION	It looks like the liquidity logic will leave some Matic dust sometimes when it runs under the right conditions.
RECOMMENDATION	Add a function to withdraw that Matic. Add logic to buy back the SouthPark token and burn it. Add logic to distribute the Matic to SouthPark Holders.
MITIGATED/COMMENT	The project team has acknowledged the issue and due to the fact that the contract is already deployed and non-upgradable, the project team determined the code to be left unchanged

Third-Party Reliance

SEVERITY	Informational
LOCATION	SouthParkToken.sol#45
RESOLVED	**No**

```
constructor(
    address _operator,
    IUniswapV2Router02 _router
  ) OwnableToken("South Park Token", "KENNY") {
    operator = _operator;
    router = _router;
  }
```

DESCRIPTION	The contract functionality depends on the reliability of a 3rd party service, in this case, the Pancakeswap router.
RECOMMENDATION	Have control over the router value, so that it can be changed if the 3rd party upgrades their router. Make sure to monitor the activity on the 3rd party router as the core logic depends on it.
MITIGATED/COMMENT	The project team has acknowledged the issue and due to the fact that the contract is already deployed and non-upgradable, the project team determined the code to be left unchanged

Owner Bypass Liquidity Rules

SEVERITY	Medium
LOCATION	SouthParkToken.sol#68
RESOLVED	**Yes**

```
function _transfer(address sender, address recipient, uint256 amount) internal virtual override {
    if (autoLiqEnabled && sender != owner() && sender != kennyEthPair) {
      autoLiq();
    }

    if (transferTaxRate > 0 && kennyEthPair != address(0)) {
      uint taxAmount = amount * transferTaxRate / 10000;
      uint rewardAmount = taxAmount * taxRewardPartRate / 100;
      uint autoLiqAmount = taxAmount - rewardAmount;

      // transfer for auto liq
      super._transfer(sender, address(this), autoLiqAmount);

      // transfer reward to masterchef
      _approve(address(this), owner(), rewardAmount);
      super._transfer(sender, owner(), rewardAmount);
      IHaveReinvestPool(owner()).addReinvestReward(rewardAmount);

      amount -= taxAmount;
    }

    super._transfer(sender, recipient, amount);
  }
```

DESCRIPTION	The owner can transfer without being included in the liquidity logic.
RECOMMENDATION	Make sure the owner of the contract is a Timelock
MITIGATED/COMMENT	On-chain analysis showed that MasterChef is the owner, the issue is thereby resolved.

Mintable And Burnable

SEVERITY	High
LOCATION	OwnableToken.sol#10
RESOLVED	**Yes**

```
1  function mint(address account, uint amount) public onlyOwner {
2    _mint(account, amount);
3  }
4
5  function burn(address account, uint amount) public onlyOwner {
6    _burn(account, amount);
7  }
```

DESCRIPTION	SouthParkToken Owner can mint and burn
RECOMMENDATION	Make sure the ownership of the contract is controlled by a governance system or at the very least by a Timelock.
MITIGATED/COMMENT	On-chain analysis showed that MasterChef is the owner so the issue is resolved.

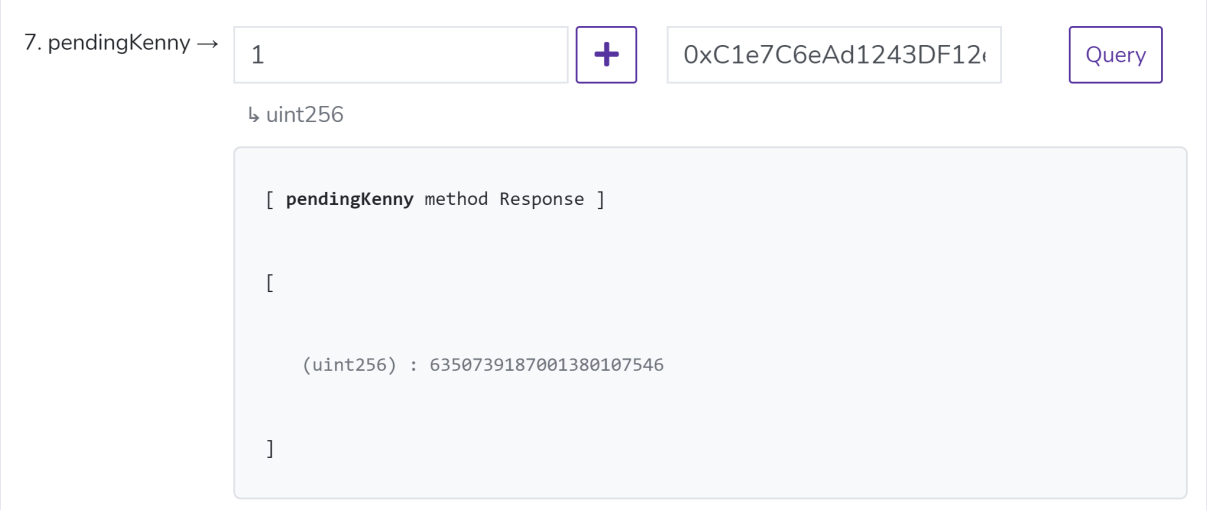
Static Analysis

No Findings

On-Chain Analysis

Operator Staking Centralization

SEVERITY	Informational
LOCATION	Masterchef
RESOLVED	**No**



Note: *this corresponds to 6350 unclaimed kenny on the corresponding pool*

DESCRIPTION	The token operator(at the moment of writing **0xC1e7C6eAd1243DF12e9D8B51b0706690b44D83c6**) is using the pools to stake funds and receive more rewards. The operator also controls a lot of lp tokens.
RECOMMENDATION	Make sure users understand how much of the supply the operator account owns.
MITIGATED/COMMENT	

Appendix A - Reviewed Documents

Document	**Address**
SouthParkToken.sol	0x6b1faaa2771E8B3AA0e0ba6830436E2DF2a0abD6
Masterchef.sol	0xbC44c49A159a0a74BC27FdAfB28EA6C70D9eb9c2
Timelock.sol	0x6a842e9d76D738f8D9DcBA2Fd31AB6824cd9F8E6
OwnableToken.sol	0x6b1faaa2771E8B3AA0e0ba6830436E2DF2a0abD6

Appendix B - Risk Ratings

Risk	Description
High Risk	A fatal vulnerability that can cause immediate loss of Tokens / Funds
Medium Risk	A vulnerability that can cause some loss of Tokens / Funds
Low Risk	A vulnerability that can be mitigated
Informational	No vulnerability

Appendix C - Icons

Icon	Explanation
	Solved by Project Team
	Under Investigation of Project Team
	Unsolved

Appendix D - Testing Standard

An ordinary audit is conducted using these steps.

1. Gather all information
2. Conduct a first visual inspection of documents and contracts
3. Go through all functions of the contract manually (2 independent auditors)
    a. Discuss findings
4. Use specialized tools to find security flaws
    a. Discuss findings
5. Follow up with project lead of findings
6. If there are flaws, and they are corrected, restart from step 2
7. Write and publish a report

During our audit, a thorough investigation has been conducted employing both automated analysis and manual inspection techniques. Our auditing method lays a particular focus on the following important concepts:

- Ensuring that the code and codebase use best practices, industry standards, and available libraries.
- Testing the contract from different angles ensuring that it works under a multitude of circumstances.
- Analyzing the contracts through databases of common security flaws.

OBELISK

Part of Tibereum Group