Kommunitas LP Farm

Smart Contract Security Audit

Prepared by ShellBoxes
Feb 2nd, 2024 - Feb 7th, 2024
Shellboxes.com
contact@shellboxes.com

Document Properties

Client	Kommunitas
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
`https://github.com/Kommunitas-net/KommunitasStakingLP`	f246f1a6b7cb8e3842001d29139fd474b60bafea

Re-Audit

Repository	Commit Hash
`https://github.com/Kommunitas-net/KommunitasStakingLP`	422e4de53525374a88eae2c4caf988c9a4243c7a

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Kommunitas engaged ShellBoxes to conduct a security assessment on the Kommunitas LP Farm beginning on Feb 2nd, 2024 and ending Feb 7th, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Kommunitas

The LP Staking Program at Kommunitas stands as a pivotal component within their decentralized crowdfunding framework. This initiative, designed to enhance liquidity and incentivize participation, represents a significant evolution in community-driven projects.

Issuer	Kommunitas
Website	`https://www.kommunitas.net`
Type	Solidity Smart Contract
Documentation	kommunitas Docs
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Kommunitas LP Farm implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 1 critical-severity, 1 high-severity, 2 medium-severity, 3 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Centralization in Rewards Distribution Mechanism	CRITICAL	Mitigated
SHB.2. Risk of Token Loss and Desynchronization	HIGH	Fixed
SHB.3. Front Run Attack	MEDIUM	Fixed
SHB.4. Handling Deflationary Tokens in Stake Function	MEDIUM	Fixed
SHB.5. Floating Pragma	LOW	Fixed
SHB.6. Missing payment Address Verification	LOW	Fixed
SHB.7. Missing Validation for Unstake Duration Parameter	LOW	Fixed

3 Finding Details

SHB.1 Centralization in Rewards Distribution Mechanism

- Severity : CRITICAL
- Status : Mitigated
- Likelihood : 3
- Impact : 3

Description:

The project's approach to distributing rewards, as described, involves manual calculation and allocation based on snapshots. This method introduces a significant level of centralization into the rewards mechanism, relying on the project owner or administrators to determine and distribute rewards. Such a centralized approach can lead to several issues, including potential bias, errors in reward calculation, delays in distribution, and a lack of transparency and trust from the stakeholders. In decentralized finance (DeFi) and blockchain projects, the expectation is typically for operations, especially critical ones like rewards distribution, to be automated and trustless, leveraging smart contracts to ensure fairness, transparency, and security.

Files Affected:

SHB.1.1: KommunitasStakingLP.sol

```
    function claim() external {
        StakeInfo storage stakerInfo = stakes[msg.sender];
        require(
            block.timestamp > stakerInfo.claimableEpoch &&
                stakerInfo.claimableEpoch > 0,
            "Unstake time is not reached yet"
        );
        require(stakerInfo.amount > 0, "No staked amount to claim");

```

```
        IERC20(payment).safeTransfer(msg.sender, stakerInfo.amount);

        _removeStaker(msg.sender);

        emit Claim(msg.sender, stakerInfo.amount, block.timestamp);
    }
```

Recommendation:

To mitigate the risks associated with centralized rewards distribution and align with the principles of decentralization in blockchain projects, it's recommended to automate the rewards mechanism through smart contracts.

- Automated Reward Calculation: Implement smart contract functions that calculate rewards based on predefined criteria such as staking duration, amount staked. These calculations should be transparent and verifiable by anyone to ensure trust.
- On-Chain Reward Allocation: Design the system to automatically allocate rewards to users' addresses based on the calculated amounts. This allocation should happen within the blockchain environment without the need for manual intervention.

Updates

The team has mitigated the risk by implementing a snapshot mechanism in the KommunitasStakingLp contract. This mechanism records stakers' information each time the snapshot function is triggered by the snapshoter at a specific time. This function calculates users' stakes and their locked tokens. For more details, the Kommunitas team has already provided documentation about the Monthly Millionaire Partner Sharing snapshot and the Stakers Rewards. In this case, rewards will be manually handled based on the state records stored in the snapshots.

SHB.1.2: KommunitasStakingLP.sol

```
    function snapshot() external onlySnapshoter {
        uint120 snapshotTime = uint120(block.timestamp);
        uint256 counter = 0;
        for (uint256 i = 0; i < this.getStakerCount(); ++i) {
```

```
            StakeInfo storage stakerInfo = stakes[stakers[i]];
            // set snapshot
            if (stakerInfo.claimableEpoch == 0) {
                SnapshotInfo memory snapshotInfo = SnapshotInfo(
                    stakers[i],
                    stakerInfo.amount
                );
                snapshots[snapshotTime].push(snapshotInfo);
                counter++;
            }
        }
        SnapshotPeriod memory sPeriod = SnapshotPeriod(
            snapshotTime,
            uint8(counter),
            totalStakedAmount
        );
        snapshotPeriod.push(sPeriod);
    }
```

SHB.2 Risk of Token Loss and Desynchronization

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

The KommunitasStakingLP contract poses a risk of token locking and potential desynchronization due to its reliance on a specific global token address (receiver address) for stake, unstake, and claim token functionalities. While the contract tracks the staked amount of each user for the global payment address, it allows the contract owner to update this address using the updatePayment function.

This flexibility introduces the risk of inadvertently locking all user tokens and creating a desynchronization between the staked tokens in the contract and any new tokens that may be introduced.

Files Affected:

SHB.2.1: KommunitasStakingLP.sol

```
struct StakeInfo {
    uint256 amount;
    uint256 claimableEpoch;
    uint256 index;
}
```

SHB.2.2: KommunitasStakingLP.sol

```
mapping(address => StakeInfo) public stakes;
address[] public stakers;
```

SHB.2.3: KommunitasStakingLP.sol

```
function updatePayment(address _payment) external onlyOwner {
    payment = _payment;
}
```

Recommendation:

To mitigate this risk, it is crucial to ensure coherence and consistency in the contract logic by establishing a fixed and immutable token address within the contract. This can be achieved by initializing the token address as a constant variable in the contract code. Additionally, any functions that allow the owner to modify this token address, such as updatePayment, should be removed to prevent the possibility of inadvertently locking user tokens or causing desynchronization issues.

Updates

The team has addressed the issue by removing the updatePayment function and implementing immutability for the tokenAddress variable within the contract.

Consequently, the tokenAddress is now initialized solely during contract deployment, with no provision for modification through any function.

SHB.2.4: KommunitasStakingLP.sol

```
    address public immutable tokenAddress;
```

SHB.3 Front Run Attack

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The contract owner can update the unstakeDuration variable using the updateUnstakeDuration function. This presents a vulnerability where the owner can front-run user unstake transactions, potentially manipulating the claimableEpoch in their StakeInfo. This issue allows the owner to preemptively adjust the unstake duration during user transactions, compromising the fairness and transparency of the stake structure.

Files Affected:

SHB.3.1: KommunitasStakingLP.sol

```
    function updateUnstakeDuration(
        uint256 _unstakeDuration
    ) external onlyOwner {
        unstakeDuration = _unstakeDuration;
    }
```

Recommendation:

To mitigate this risk, we propose the following solutions:

- Ensure that user unstake transactions can validate the actual unstake duration by adding an expectedUnstakeDuration parameter to the function parameters. Validate that this value should be equal to the actual unstakeDuration contract variable. This verification will prevent the owner from manipulating the unstake duration during user transactions.

- Alternatively, add an attribute named unstakeDuration in the StakeInfo struct, which will be initialized using the stake transaction. This approach ensures that the unstake duration is recorded at the time of staking, preventing manipulation by the contract owner during unstake transactions. And, include the expectedDuration parameter in the stake parameters to prevent front-running during staking actions.

- Or, initialize the unstakeDuration in the contract constructor and remove the update-UnstakeDuration function.

Implementing either of these solutions enhances the fairness and transparency of the contract's stake structure, reducing the risk of front-running attacks and ensuring a more equitable user experience.

Updates

The team has addressed the issue by adding the _expectedUnstakeDuration parameter in the unstake function and ensuring validation that this value matches the actual unstakeDuration contract variable.

SHB.3.2: KommunitasStakingLP.sol

```
    function unstake(uint120 _expectedUnstakeDuration) external {
        StakeInfo storage stakerInfo = stakes[msg.sender];
        require(
            unstakeDuration == _expectedUnstakeDuration,
            "Unstake duration is not matched!"
        );
```

SHB.4 Handling Deflationary Tokens in Stake Function

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The stake function allows users to stake tokens by transferring them from the user's address to the contract. However, it does not account for the potential impact of deflationary tokens, which automatically reduce the amount transferred as a fee or burn a portion of the transaction. This oversight could lead to discrepancies between the amount intended to be staked by the user and the amount actually received by the contract, affecting the accuracy of staking records and user balances.

Files Affected:

SHB.4.1: KommunitasStakingLP.sol

```
    function stake(uint256 _amount) external {
        require(_amount > 0, "Stake amount must be greater than zero");

        StakeInfo storage stakerInfo = stakes[msg.sender];

        if (stakerInfo.amount == 0) {
            stakers.push(msg.sender);
            stakerInfo.index = stakers.length - 1;
        }

        stakerInfo.amount += _amount;
        stakerInfo.claimableEpoch = 0;

        IERC20(payment).safeTransferFrom(msg.sender, address(this),
            ↪ _amount);
```

```

        emit Stake(msg.sender, _amount);
    }
```

Recommendation:

To address the issue of handling deflationary tokens, it's recommended to verify the actual transferred amount and update the staker's information accordingly. This can be done by checking the contract's balance of the token before and after the transfer, rather than relying on the _amount parameter directly.

Updates

The team has resolved the issue by checking the contract's balance of the token before and after the transfer and updating the staker info amount (stakerInfo.amount) with the actual transferred amount to the contract.

SHB.4.2: KommunitasStakingLP.sol

```
    function stake(uint128 _amount) external {
        require(_amount > 0, "Stake amount must be greater than zero");

        // calculate real amount transfered to the contract
        uint256 balanceBefore = IERC20(tokenAddress).balanceOf(address(
            ↪ this));
        IERC20(tokenAddress).safeTransferFrom(
            msg.sender,
            address(this),
            _amount
        );
        uint256 balanceAfter = IERC20(tokenAddress).balanceOf(address(
            ↪ this));

        // set _amount based on real amount transferred and then process
            ↪ stake
        _amount = uint128(balanceAfter - balanceBefore);
```

```
        StakeInfo storage stakerInfo = stakes[msg.sender];

        if (stakerInfo.amount == 0) {
            stakers.push(msg.sender);
            stakerInfo.index = uint8(stakers.length) - 1;
        }

        stakerInfo.amount += _amount;
        stakerInfo.claimableEpoch = 0;
        totalStakedAmount += _amount;

        emit Stake(msg.sender, _amount);
    }
```

SHB.5 Floating Pragma

- Severity: LOW
- Status: Fixed
- Likelihood: 1
- Impact: 2

Description:

The KommunitasStakingLP contract uses a floating Solidity pragma of 0.8.23, indicating compatibility with any compiler version from 0.8.23 (inclusive) up to, but not including, version 0.9.0. This flexibility could potentially introduce unexpected behavior if the contracts are compiled with a newer compiler version that includes breaking changes.

Files Affected:

SHB.5.1: KommunitasStakingLP.sol

```
pragma solidity ^0.8.23;
```

Recommendation:

It is generally recommended to lock the pragma statement to a specific Solidity compiler version to ensure consistent behavior across different compiler versions. To achieve this, consider removing the caret (^) from the pragma statement and specifying a fixed version, such as pragma solidity 0.8.23.

Updates

The team has resolved this issue by fixing the pragma version of the KommunitasStakingLP contract, locking it to 0.8.23.

SHB.6 Missing payment Address Verification

- Severity : LOW
- Likelihood : 1
- Status : Fixed
- Impact : 2

Description:

The contract constructor and the updatePayment function lacks a critical address verification check and allows the payment to be set to any address, including address(0). This absence of address validation poses a potential risk, as setting the payment to address(0) may block all contract staking features.

Files Affected:

SHB.6.1: KommunitasStakingLP.sol

```
constructor(uint256 _unstakeDuration, address _payment) {
    owner = msg.sender;
    unstakeDuration = _unstakeDuration;
    payment = _payment;
}
```

SHB.6.2: KommunitasStakingLP.sol

```
    function updatePayment(address _payment) external onlyOwner {
        payment = _payment;
    }
```

Recommendation:

To mitigate this issue, it is essential to incorporate a check in the constructor and updatePayment function to validate that the _payment address is not address(0).

By implementing these checks, the contract can prevent critical functions from being disabled due to incorrect or malicious address inputs, enhancing overall security and robustness.

Updates

The team has resolved the issue by adding a zero address check on the _tokenAddress variable upon initialization in the constructor and removing the updatePayment function.

SHB.6.3: KommunitasStakingLP.sol

```
    constructor(
        address _tokenAddress,
        uint120 _minUnstakeDuration,
        uint120 _maxUnstakeDuration,
        uint120 _unstakeDuration
    ) {
        require(_tokenAddress != address(0), "Invalid token address");
```

SHB.7 Missing Validation for Unstake Duration Parameter

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The smart contract's constructor and the updateUnstakeDuration function both set the unstakeDuration parameter without any validation checks. This absence of validation could potentially allow setting a duration that is either too short or too long (which could lock users' funds for an impractical amount of time). Depending on the intended functionality and security requirements of the contract.

Files Affected:

SHB.7.1: KommunitasStakingLP.sol

```
constructor(uint256 _unstakeDuration, address _payment) {
    owner = msg.sender;
    unstakeDuration = _unstakeDuration;
    payment = _payment;
}
```

SHB.7.2: KommunitasStakingLP.sol

```
function updateUnstakeDuration(
    uint256 _unstakeDuration
) external onlyOwner {
    unstakeDuration = _unstakeDuration;
}
```

Recommendation:

To mitigate these risks, it is recommended to introduce validation checks for the unstakeDuration parameter in both the constructor and the updateUnstakeDuration function. These checks should ensure that the unstake duration is within reasonable and secure bounds.

Updates

The team has addressed the issue by adding minUnstakeDuration and maxUnstakeDuration variables and implementing verification to ensure that the unstakeDuration remains within the specified bounds

SHB.7.3: KommunitasStakingLP.sol

```
        require(
            _unstakeDuration >= _minUnstakeDuration &&
                _unstakeDuration <= _maxUnstakeDuration,
            "Unstake duration is out of bound"
        );
```

SHB.7.4: KommunitasStakingLP.sol

```
    function updateUnstakeDuration(
        uint120 _unstakeDuration
    ) external onlyOwner {
        require(
            _unstakeDuration >= minUnstakeDuration &&
                _unstakeDuration <= maxUnstakeDuration,
            "Unstake duration is out of bound"
        );
        unstakeDuration = _unstakeDuration;
    }
```

4 Best Practices

BP.1 Gas-Efficient Struct Packing

Description:

Optimizing storage for the StakeInfo struct can significantly enhance gas efficiency by utilizing tighter packing of its variables. The original struct definition allocates more storage space than necessary for each variable. By repacking the variables, we can reduce the storage footprint of the struct while maintaining the integrity of the data. For instance, converting uint256 variables to smaller data types like uint128 and uint8 can effectively reduce gas costs associated with storage operations.

BP.1.1: KommunitasStakingLP

```
struct StakeInfo {
uint128 amount;
uint120 claimableEpoch;
uint8 index;
}
```

By repacking the struct variables in this manner, we optimize storage usage and improve gas efficiency, resulting in cost savings for contract interactions.

Files Affected:

BP.1.2: KommunitasStakingLP.sol

```
    struct StakeInfo {
        uint256 amount;
        uint256 claimableEpoch;
        uint256 index;
    }
```

Status - Fixed

BP.2 Rename Contract Variables

Description:

The KommunitasStakingLP contract implements a two-step owner update process using a proposed owner address first, which is set into the receiver variable in the contract. Subsequently, this receiver should confirm and call the updateOwner function to become the new owner. We recommend renaming this receiver variable to a more accurate name reflecting its purpose, such as proposedOwner. Additionally, the token address variable in this staking contract, named payment, may lead to confusion. We recommend renaming it to tokenAddress or with the token name, for example, KOM_TOKEN. This will ensure transparency and clear understanding of its purpose, enhancing readability and maintainability of the contract code.

Files Affected:

BP.2.1: KommunitasStakingLP.sol

```
    address public receiver;
    address public payment;
```

Status - Fixed

5 Conclusion

We examined the design and implementation of Kommunitas LP Farm in this audit and found several issues of various severities. We advise Kommunitas team to implement the recommendations contained in all 7 of our findings to further enhance the code's security. It is of utmost priority to start by addressing the most severe exploit discovered by the auditors then followed by the remaining exploits, and finally we will be conducting a re-audit following the implementation of the remediation plan contained in this report.

We would much appreciate any constructive feedback or suggestions regarding our methodology, audit findings, or potential scope gaps in this report.

6 Scope Files

6.1 Audit

Files	MD5 Hash
KommunitasStakingLP.sol	94f27e6a9a112dcb78f8c259c1395bd5

6.2 Re-Audit

Files	MD5 Hash
KommunitasStakingLP.sol	5fa168acc1fe9779d250343772a03066

7 Disclaimer

Shellboxes reports should not be construed as "endorsements" or "disapprovals" of particular teams or projects. These reports do not reflect the economics or value of any "product" or "asset" produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com