Nimbi Wolfpack Audit Report

Prepared by: @zarkk01

Table of Contents

About

Nimbi is a decentralized eco-system which operates on the ERC-404 token protocol aiming to build a Web3 game (Crypto Dust Runner) and other innovations. Also, it supports the staking of its native token and a presale mechanism that will further contribute to the growth of the Nimbi ecosystem.

@zarkk01 is an independent blockchain security researcher having experience in auditing smart contracts and DeFi protocols and secured multiple projects in the past. The audit was conducted to ensure the security of the Nimbi Wolfpack ecosystem and to identify potential vulnerabilities in the smart contracts.

Disclaimer

@zarkk01 makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts. A smart contract security review can never verify the complete absence of vulnerabilities.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L

	Impact			
	Low	M	M/L	L

Audit Details

Dates : `6-10 November 2024`

Review commit hash : `dbe8b5b3aec9935b20af4e1dcbb56db7984be05c`

Scope

The following smart contracts were in scope of the audit:

- `Kazi.sol` (ERC20 Implementation, Bugs found : 0)
- `nimbi.sol` (ERC404 Implementation, Bugs found : 0)
- `Presale.sol` (Presale smart contract, Bugs found : 6)
- `Staking.sol` (Staking smart contract, Bugs found : 0)
- `ERC404.sol`

Executive Summary

Severity	No. of Issues
High	4
Medium	4
Low	3

Issues found

Findings

High

[H-01] Rewards are not zeroed out after being claimed leading to the draining of `Staking` contract.

Severity

Impact: High **Likelihood:** High

Description

The `claimReward()` function in the `Staking` contract doesn't reset the user's reward balance to zero after the rewards are claimed. As a result, users can repeatedly call this function and drain the `Staking` contract by claiming the same reward amount *multiple* times. The exploitability is high because any user who has earned rewards can trigger this flaw without restrictions.

```
    function claimReward() external updateReward(msg.sender) {
        uint256 reward = s_rewards[msg.sender];
        bool success = s_rewardToken.transfer(msg.sender, reward);
        if (!success) {
            revert Staking__TransferFailed();
        }
    }
```

Recommendation

Add a line to zero out the rewards after they have been claimed :

```
    function claimReward() external updateReward(msg.sender) {
        uint256 reward = s_rewards[msg.sender];
+       s_rewards[msg.sender] = 0;
        bool success = s_rewardToken.transfer(msg.sender, reward);
        if (!success) {
            revert Staking__TransferFailed();
        }
    }
```

[H-02] `Staking::updateReward` function incorrectly attach the `block.timestamp` to the `s_balance` messing up the accountings of the contract.

Severity

Impact: High **Likelihood:** High

Description

The `updateReward` modifier in the `Staking` contract mistakenly assigns `block.timestamp` to the `s_balances` mapping, which is intended to store user staking balances. This incorrect use of `s_balances` for time-tracking messes up the reward calculation logic, as balance data is overwritten with timestamps. This leads to catastrophic results since the reward distributions will be totally wrong. The likelihood of this issue being exploited is high, as any user interaction that interact with the contract will encounter this flaw.

```
    modifier updateReward(address account) {
        s_rewards[account] = earned(account);
@>          s_balances[account] = block.timestamp;
        _;
    }
```

Recommendation

Introduce a new mapping to store the last time the reward was updated and use it to calculate the rewards :

```
+   mapping(address => uint256) s_lastUpdateTime;

    modifier updateReward(address account) {
        s_rewards[account] = earned(account);
+       s_lastUpdateTime[account] = block.timestamp;
        _;
    }

    function earned(address account) public view returns (uint256) {
        uint256 stakedAmount = s_balances[account];
-        uint256 lastRewardTime = s_balances[account];
+        uint256 lastRewardTime = s_lastTimeUpdate[account];
        uint256 stakedDuration = block.timestamp - lastRewardTime;

        uint256 annualReward = (stakedAmount * ANNUAL_REWARD_PERCENTAGE) /
100;
        uint256 earnedReward = (annualReward * stakedDuration) /
SECONDS_IN_A_YEAR;

        return  s_rewards[account] + earnedReward;
    }
```

[H-03] `Presale::claim()` calls will always revert for every user since `vestingUnfreezeTime[user][_id]` is always 0.

Severity

Impact: High **Likelihood:** Low

Description

During `Presale::claim()` function, the contract checks if the `vestingUnfreezeTime[user][_id]` is greater than 0. If it is not, the function will revert. However, the `vestingUnfreezeTime[user][_id]` is never initialized in the contract, so it will always be 0. This means that the `claim()` function will always revert for every user.

```
    function claim(address user, uint256 _id) public returns (bool) {
        Vesting memory _userVesting = userVesting[user][_id];
        require(_userVesting.totalAmount > 0, "Nothing to claim");

        // Check if vesting has been unfrozen
@>        require(vestingUnfreezeTime[user][_id] > 0, "Tokens are
frozen");

        // ...
    }
```

Recommendation

It is recommended to initialize the `vestingUnfreezeTime[user][_id]` on the `buy` functions :

```
        if (userVesting[_msgSender()][_id].totalAmount > 0) {
            userVesting[_msgSender()][_id].totalAmount += (amount *
                _presale.baseDecimals);
        } else {
            userVesting[_msgSender()][_id] = Vesting(
                (amount * _presale.baseDecimals),
                0,
                _presale.vestingStartTime + _presale.vestingCliff,
                _presale.vestingStartTime +
                    _presale.vestingCliff +
                    _presale.vestingPeriod
            );
+             vestingUnfreezeTime[_msgSender()][_id] =
_presale.vestingStartTime + _presale.vestingCliff;
        }
```

[H-04] `Presale::claim()` will revert due to underflow if user has claimed more tokens than what he is about claim this time.

Severity

Impact: High **Likelihood:** High

Description

`Presale::claimableAmount()` function is called every time someone tries to claim their vested tokens in `Presale::claim()`. This function is supposed to calculate the amount of tokens that can be claimed by the user. This amount is 10% of the total vested tokens per claim (every 2 weeks). However, the calculation of the amount to claim is incorrect as it is always subtracting the `_user.claimedAmount`. The `(noOfClaims * perClaimAmount)` part is right, since it indeed calculates how many weeks have been passed since the last claim and multiplies it by 10% of the total vested tokens. However, there is no reason to subtract the `_user.claimedAmount` from this amount and it will deflate the reward amounts until it start reverting due to underflow. The required check to ensure that the user has not claimed more tokens than what he is about to claim this time can be added after the calculation of `amountToClaim` (look at Recommendation). Let's consider the following scenario:

1. User has 1000 tokens vested.
2. User has claimed 300 tokens in 3 claims.
3. User tries to claim his tokens for the 4th time which is 10% of the total vested tokens (1000 * 10% = 100).
4. The `amountToClaim = (noOfClaims * perClaimAmount) - _user.claimedAmount;` part will revert due to underflow since `noOfClaims` is 1, `perClaimAmount` is 100 and `_user.claimedAmount` is 300.

```
    function claimableAmount(
        address user,
        uint256 _id
    ) public view checkPresaleId(_id) returns (uint256) {
        Vesting memory _user = userVesting[user][_id];
        require(_user.totalAmount > 0, "Nothing to claim");

        uint256 amount = _user.totalAmount - _user.claimedAmount;
        require(amount > 0, "Already claimed");

        if (block.timestamp < _user.claimStart) return 0;

        uint256 noOfClaims = (block.timestamp -
            vestingUnfreezeTime[user][_id]) / (2 * 7 * 24 * 3600);
        uint256 perClaimAmount = (_user.totalAmount * 10) / 100; // 10%
per claim

@>        uint256 amountToClaim = (noOfClaims * perClaimAmount) -
            _user.claimedAmount;

        return amountToClaim;
    }
```

Recommendation

It is recommended to remove the `- _user.claimedAmount` part from the `claimableAmount()` function and add the following check after this to ensure that the user will not claim more than intended :

```
    function claimableAmount(
        address user,
        uint256 _id
    ) public view checkPresaleId(_id) returns (uint256) {
        Vesting memory _user = userVesting[user][_id];
        require(_user.totalAmount > 0, "Nothing to claim");

        uint256 amount = _user.totalAmount - _user.claimedAmount;
        require(amount > 0, "Already claimed");

        if (block.timestamp < _user.claimStart) return 0;

        uint256 noOfClaims = (block.timestamp -
            vestingUnfreezeTime[user][_id]) / (2 * 7 * 24 * 3600);
        uint256 perClaimAmount = (_user.totalAmount * 10) / 100; // 10%
per claim

-        uint256 amountToClaim = (noOfClaims * perClaimAmount) -
-            _user.claimedAmount;

+        uint256 amountToClaim = (noOfClaims * perClaimAmount);
+        require(amountToClaim <= amount, "Claimed amount is more than
what has remained to be claimed");

        return amountToClaim;
    }
```

Medium

[M-01] Buyer can put his own address as a referrer and get a 10% boost instantly.

Severity

Impact: Medium **Likelihood:** Medium

Description

Any buyer can put his own address as a referrer and get a 10% boost instantly. This can be done by calling the `buyWithUSDT` function in the `Presale` contract and passing the buyer's address as the referrer address. In this way, the buyer can get a 10% reward without any actual referral bypassing the referral system and intentions of the ecosystem.

```
    function buyWithUSDT(
        uint256 _id,
        uint256 amount,
        address referrer
    ) external checkPresaleId(_id) checkSaleState(_id, amount) returns
(bool) {
        // ...

        // Handle referral reward
        if (referrer != address(0) && referral[_msgSender()] ==
address(0)) {
            referral[_msgSender()] = referrer;
@>            uint256 referralReward = (amount * 10) / 100; // 10% reward

            if (referralReward > 0) {
@>                userVesting[referrer][_id].totalAmount +=
                    referralReward *
                    _presale.baseDecimals;
                hasClaimedReferral[_msgSender()] = true;
                emit ReferralRewarded(referrer, _msgSender(),
referralReward);
            }
        }

        // ...
    }
```

Recommendation

It is recommended to check if the referral address is the same as the buyer's address :

```
    function buyWithUSDT(
        uint256 _id,
        uint256 amount,
        address referrer
    ) external checkPresaleId(_id) checkSaleState(_id, amount) returns
(bool) {
        // ...

        // Handle referral reward
-        if (referrer != address(0) && referral[_msgSender()] ==
address(0)) {
+        if (referrer != address(0) && referral[_msgSender()] ==
address(0) && referrer != _msgSender()) {
            referral[_msgSender()] = referrer;
            uint256 referralReward = (amount * 10) / 100; // 10% reward

            if (referralReward > 0) {
                userVesting[referrer][_id].totalAmount +=
                    referralReward *
                    _presale.baseDecimals;
                hasClaimedReferral[_msgSender()] = true;
                emit ReferralRewarded(referrer, _msgSender(),
referralReward);
            }
        }

        // ...
    }
```

[M-02] During `Presale::createPresale()`, the `saleToken` of every presale is set as `address(0)` leading to reversion of every `claim()`.

Severity

Impact: High **Likelihood:** Medium

Description

During the `createPresale` function, the `saleToken` of every presale is set as `address(0)`. This leads to the reversion of every `claim()` function call as the `IERC20(presale[_id].saleToken).transfer(user, amountToClaim)` call will revert due to the `saleToken` being set as `address(0)`. However, this is possible and **must** change before the claim period starts by `Presale::changeSaleTokenAddress()`.

```
    function createPresale(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _price,
        uint256 _tokensToSell,
        uint256 _baseDecimals,
        uint256 _vestingStartTime,
        uint256 _vestingCliff,
        uint256 _vestingPeriod,
        uint256 _enableBuyWithEth,
        uint256 _enableBuyWithUsdt
    ) external onlyOwner {
        // ...
        presale[presaleId] = Presale(
@>              address(0),
            _startTime,
            _endTime,
            _price,
            _tokensToSell,
            _baseDecimals,
            _tokensToSell,
            _vestingStartTime,
            _vestingCliff,
            _vestingPeriod,
            _enableBuyWithEth,
            _enableBuyWithUsdt
        );
        // ...
    }

    function claim(address user, uint256 _id) public returns (bool) {
        // ...
        userVesting[user][_id].claimedAmount += amountToClaim;
@>        bool status = IERC20(presale[_id].saleToken).transfer(
            user,
            amountToClaim
        );
        require(status, "Token transfer failed");

        // ...
    }
```

Recommendation

It is recommended to put a check in the `createPresale` function to ensure that the `saleToken` is not set as `address(0)` and instead it is the address of the token being sold in the presale.

[M-03] Lack of `stale` price check in `Presale::getLatestPrice()` function.

Severity

Impact: High **Likelihood:** Low

Description

As per Chainlick documentation, it is recommended to **always** check the timestamp of the latest price update to ensure that the price is not stale. It is possible that the price feed is not updated for a long time and the price is outdated and, as a result, the contract will retrieve the outdated price. This can lead to a wrong calculation of the token price and can cause a loss to the users.

```
    function getLatestPrice() public view returns (uint256) {
@>        (, int256 price, , , ) = aggregatorInterface.latestRoundData();
        price = (price * (10 ** 10));
        return uint256(price);
    }
```

Recommendation

It is recommended to put a check for stale price updates from Chainlink feed :

```
    function getLatestPrice() public view returns (uint256) {
-        (, int256 price, , , ) = aggregatorInterface.latestRoundData();
+        (, int256 price, , uint256 updateAt , ) =
aggregatorInterface.latestRoundData();
+        require(updateAt < block.timestamp - 1 hours, "Price is
outdated");
        price = (price * (10 ** 10));
        return uint256(price);
    }
```

[M-04] `referrer` will not be able to claim his referral rewards if he is not a buyer for the presale himself.

Severity

Impact: High **Likelihood:** Medium

Description

When someone puts a `referrer` so to get a 10% reward, the `buyWithEth` and `buyWithUsdt` functions does not ensure that there is a `Vesting` struct for the `referrer`, they just increase its `totalAmount`. However, if the `referrer` has not bought himself tokens during this presale, he will not have populate his `userVesting` mapping and there will be no `claimStart` and `claimEnd`. This will have as a result that the `referrer` will not be able to claim his referral rewards since the `claimableAmount()` function checks the `claimStart` of the user claiming.

```
    function claimableAmount(
        address user,
        uint256 _id
    ) public view checkPresaleId(_id) returns (uint256) {
        // ...

@>        if (block.timestamp < _user.claimStart) return 0;

        uint256 noOfClaims = (block.timestamp -
            vestingUnfreezeTime[user][_id]) / (2 * 7 * 24 * 3600);
        uint256 perClaimAmount = (_user.totalAmount * 10) / 100; // 10%
per claim

        // ...
    }
```

```
    function buyWithUSDT(
        uint256 _id,
        uint256 amount,
        address referrer
    ) external checkPresaleId(_id) checkSaleState(_id, amount) returns
(bool) {
        // ...

        // Handle referral reward
        if (referrer != address(0) && referral[_msgSender()] ==
address(0)) {
```

```
            referral[_msgSender()] = referrer;
            uint256 referralReward = (amount * 10) / 100; // 10% reward

            if (referralReward > 0) {
@>                userVesting[referrer][_id].totalAmount +=
                    referralReward *
                    _presale.baseDecimals;
                hasClaimedReferral[_msgSender()] = true;
                emit ReferralRewarded(referrer, _msgSender(),
referralReward);
            }
        }

        // ...
    }
```

Recommendation

Consider checking if the `referrer` has a `Vesting` struct for this presale and if not, create one :

```
    function buyWithUSDT(
        uint256 _id,
        uint256 amount,
        address referrer
    ) external checkPresaleId(_id) checkSaleState(_id, amount) returns
(bool) {
        // ...

        // Handle referral reward
        if (referrer != address(0) && referral[_msgSender()] ==
address(0)) {
            referral[_msgSender()] = referrer;
            uint256 referralReward = (amount * 10) / 100; // 10% reward

            if (referralReward > 0) {
+                if (userVesting[referrer][_id].totalAmount > 0) {
+                    userVesting[referrer][_id].totalAmount +=
(referralReward *
+                        _presale.baseDecimals);
+                } else {
+                    userVesting[refferer][_id] = Vesting(
+                        (referralReward * _presale.baseDecimals),
+                        0,
+                        _presale.vestingStartTime +
_presale.vestingCliff,
+                        _presale.vestingStartTime +
+                            _presale.vestingCliff +
+                            _presale.vestingPeriod
+                    );
+                    vestingUnfreezeTime[referrer][_id] =
_presale.vestingStartTime + _presale.vestingCliff;
+                 }
                hasClaimedReferral[_msgSender()] = true;
                emit ReferralRewarded(referrer, _msgSender(),
referralReward);
            }
        }

        // ...
    }
```

Low

[L-01] Lack of events for staking, withdrawal, and reward claims.

Severity

Impact: Low / Informational **Likelihood:** High

Description

The `Staking` contract does not emit events for crucial actions like staking, withdrawal, or reward claiming actions, making it difficult to monitor contract activity and verify user interactions off-chain.

Recommendation

Introduce events to log staking, withdrawal, and reward claims, as shown in the code snippet below :

```
+    event Staked(address indexed user, uint256 amount);
+    event Withdrawn(address indexed user, uint256 amount);
+    event RewardClaimed(address indexed user, uint256 reward);

     function stake(uint256 amount) external updateReward(msg.sender)
moreThanZero(amount) {
         // ...
+        emit Staked(msg.sender, amount);
     }

     function withdraw(uint256 amount) external updateReward(msg.sender)
moreThanZero(amount) {
         // ...

+         emit Withdrawn(msg.sender, amount);
     }

     function claimReward() external updateReward(msg.sender) {
         // ...
+        emit RewardClaimed(msg.sender, reward);
    }
```

[L-02] Lack of `setLimit` function on `Kazi` contract.

Severity

Impact: Low **Likelihood:** Low

Description

The `Kazi` contract lacks a `setLimit` function, and as a result the owner can not **ever** update the `buyLimit` and `sellLimit` values. Without this flexibility, the contract is less adaptable to changing conditions, and this will, potentially, limit its effectiveness over time.

Recommendation

Introduce a `setLimit()` to the `Kazi` contract so to be able to make modifications if needed :

```
+    function setLimit(uint256 _buylimit, uint256 _selllimit) public
onlyOwner{
+        buyLimit = _buylimit;
+        sellLimit = _selllimit;
+    }
```

[L-03] Make sure to follow CEI pattern on `Presale::buyWithEth()` function.

Severity

Impact: Low **Likelihood:** Low

Description

To prevent re-entrancy possibilities, it is good to follow CEI (Checks - Effects - Interactions) design pattern in `Presale::buyWithEth()`.

Recommendation

Consider moving the `ETH` transfers at the end of the function :

```
    function buyWithEth(
        uint256 _id,
        uint256 amount,
```

```diff
        address referrer
    )
        external
        payable
        checkPresaleId(_id)
        checkSaleState(_id, amount)
        nonReentrant
        returns (bool)
    {
        // ...

-        sendValue(payable(owner()), ethAmount);
-        if (excess > 0) sendValue(payable(_msgSender()), excess);

        // Handle referral reward
        if (referrer != address(0) && referral[_msgSender()] ==
address(0)) {
            referral[_msgSender()] = referrer;
            uint256 referralReward = (amount * 10) / 100; // 10% reward

            if (referralReward > 0) {
                userVesting[referrer][_id].totalAmount +=
                    referralReward *
                    _presale.baseDecimals;
                hasClaimedReferral[_msgSender()] = true;
                emit ReferralRewarded(referrer, _msgSender(),
referralReward);
            }
        }

+        sendValue(payable(owner()), ethAmount);
+        if (excess > 0) sendValue(payable(_msgSender()), excess);

        // ...
    }
```