REDTEAM
Cyber Security Services

Smart Contract Security Assessment

Tokenomia.pro

Date (DD/MM/YYYY)	Task
10-29/05/2022	Security assessment
29/05/2022	Report delivery
15-21/06/2022	Retest
21/06/2022	Retest report delivery

Company Information	Contact information	
RED TEAM Sp. z o.o. Chmielna 2/31 00-020 Warsaw VAT-UE PL5252739138 https://redteam.pl info@redteam.pl	**Pawel Wylecial** *Cyber Security Expert* pawel.wylecial@redteam.pl	**Adam Ziaja** *Cyber Security Expert* adam.ziaja@redteam.pl

Revision history

Date (DD/MM/YYYY)	Version	Description
29/05/2022	1	Final report.
21/06/2022	2	Final retest report.

1. Consultants

The following cyber security consultants worked on this assignment:

- **Pawel Wylecial** – OSCP, GXPN;
- **Szymon Grzybowski**.

2. Executive summary

2.1 Description

The security assessment revealed multiple security issues which should be addressed. The most severe findings concern broken functionality which was described in business requirements related to restaking tokens and a possibility to drain staked tokens from the main staking contract. Despite the impact of the latter additional conditions have to be met to successfully exploit this issue (maintainer or owner roles), thus the finding was not marked as critical. A few findings were marked as medium as impact of potential damages for instance in presale contract or not following standard voting implementations can be significant. All details are described in the technical part of this report.

Additionally the code revealed places for potential improvements in areas like: safe funds transferring, ownership transfer handling, parameters validation, documentation, code comments and usage of emit messages for off-chain analytical tools.

2.1.1 Re-test summary

Most of the findings from the report were resolved or were accepted by the development team as in accordance with the business requirements. The most important issues were resolved.

2.2 Scope

The security assessment covered multiple Tokenomia.pro smart contracts:

- `contracts/Staking/StakingV2Balancer.sol`
- `contracts/Staking/StakingV2Mock.sol`
- `contracts/Staking/StakingV2Factory.sol`
- `contracts/Staking/StakingV2Vendor.sol`
- `contracts/Staking/StakingV2.sol`
- `contracts/Migration/Migrations.sol`
- `contracts/Token/TokenVesting.sol`
- `contracts/Token/Token.sol`
- `contracts/Token/TokenMock.sol`
- `contracts/Token/TokenStakableVesting.sol`
- `contracts/Token/AntiBot.sol`
- `contracts/Presale/SimplePresaleMock.sol`
- `contracts/Presale/SimplePresale.sol`

Smart contracts repository and commit:

- repo: `https://github.com/tokenomia-pro/tokenomia-contracts`
- commit: `d3dfcd93974c657d8bae3fabf08bc2f8cee6ef06`

Smart contracts repository and commit (retest):

- repo: `https://github.com/tokenomia-pro/tokenomia-contracts`
- commit: `a428edc69faad1ac4ce5b409e58bd409a60eae43`

2.2.1 Limitations

No extensive documentation or business requirements were provided, thus the analysis was based on the delivered implementations, litepaper and responses for questions from the development team during the audit. It is important to remember that even though the auditors try to cover and infer as much as possible from the delivered assets and understand precisely the business needs to identify vulnerabilities or general issues, in case of absence of proper business requirements in rare cases it might not be as exhaustive as expected.

The security assessment was not focused on gas optimizations, thus some areas of code might still exist where gas costs can be lowered..

2.3 Methodology

The analysis was performed using the tools widely used by Solidity developers and the community of the Ethereum and the Ethereum Virtual Machine (EVM) compatible blockchains (e.g. the Binance Smart Chain) and focused on manual inspection of the smart contract as the most effective way to uncover vulnerabilities. The toolchain includes the Remix IDE, Visual Studio Code with various Solidity plugins, for manual inspection and in dynamic analysis tools like Foundry and Ganache are incorporated. In addition common security tools for static source code analysis and fuzzing were used to spot common issues. All findings describe in detail the issue with code samples as well as the potential impact, recommendations and references if possible.

2.4 Disclaimer

The main target of the analysis was the source code of the smart contract(s) as contracts when deployed to the blockchain are publicly readable. Such an approach is called a white-box security assessment and even though it gives the best insight into the technical part of the smart contract it might not reveal all vulnerabilities and potential risks of the contract, token or the company behind the smart contract and should not be treated as a security warranty. New exploits and vulnerabilities are published regularly as well as new social engineering techniques emerge daily to convince investors to trust potentially malicious actors. Taking such disclaimer into account the review should not be considered solely as an indicator of a safe and secure investment, thus should not be seen as financial advice.

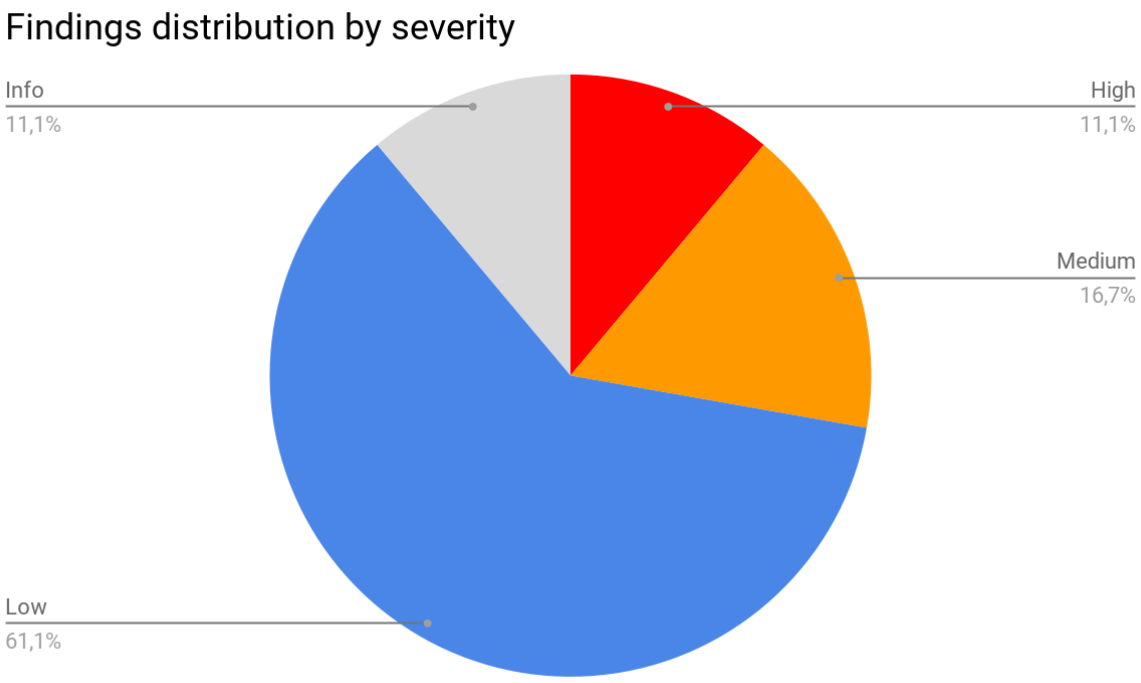
2.3 Vulnerability classification

The severity level of findings has been marked in a 5 degree scale (critical, high, medium, low, info). The actual risk for business may differ, as our evaluation is based purely on technical aspects of the vulnerability.

Severity	Description
Critical	Vulnerabilities leading to loss of funds which can be invoked by regular users without any special permissions
High	Vulnerabilities leading to loss of funds or with other important impact on the business, but requiring certain conditions to be met
Medium	Vulnerabilities or bad practices that might have a negative impact on the business
Low	Suggestions regarding best practices, optimisations or minor issues without a significant impact on the business
Info	No direct security implications

2.4 Identified vulnerabilities

No.	Severity	Title	Status
3.1	High	Nonfunctional restake functionality	Closed
3.2	High	Possibility to drain staked tokens through a malicious rewards vendor	Closed
3.3	Medium	Collected funds delivery to arbitrary address in SimplePresale	Accepted risk
3.4	Medium	Duplicates allowed in StakingV2Balancer addStakingInstance	Closed
3.5	Medium	Non standard delegate votes implementation	Accepted risk
3.6	Low	Divide by zero in StakingV2Vendor updatePool	Partially Fixed
3.7	Low	Insufficient documentation	Closed
3.8	Low	Lack of zero check for divisor in StakingV2Balancer	Closed
3.9	Low	Missing 2-step ownership transfer	Accepted risk
3.10	Low	Mixed timestamp and timespan variables in deposit function in StakingV2	Partially fixed
3.11	Low	Missing important address in emit event while adding staking to allowed list	Closed
3.12	Low	Multiple tests failing	Closed
3.13	Low	Never used variable - maxAmount	Closed
3.14	Low	No full unit test coverage	Closed
3.15	Low	No support for fee-on-charge tokens as rewards	Accepted risk
3.16	Low	Unused function parameter in StakingV2Balancer	Closed
3.17	Info	Gas optimization - external functions instead of public	Closed
3.18	Info	No emit event in StakingV2Balancer reward	Closed



3. Findings

All findings identified during the assessment are listed below. Description section explains the vulnerability and resulting threats. Technical details section contains information about location and reproduction of the finding, it is designated mostly for teams that will handle fixing the issue. Recommendations list possible remediations for the identified problem. The references section contains useful material related to the vulnerability and methods of fixing it.

3.1 Nonfunctional restake functionality [High]

Severity	Status
High	Closed

Description

The re-stake functionality was found to be not operational. The re-staking process which is used in `TokenStakableVesting.sol` as well as in `StakingV2.sol` itself which allows to move funds from a contract to another one without unnecessary fees was found to be missing spenders approval resulting in an error “insufficient allowance”.

Technical details

The process involved the following steps:

- Allow a user to add new funds to be re-staked, which are transferred to the current contract,
- Calculate the amount which should be re-staked and which are transferred elsewhere (e.g. vault) if applicable,
- Subtract the amounts from the user's balance in the current contract (`TokenStakableVesting` or `StakingV2`),
- Invoke `deposit` function of the recipient `StakingV2` contract to transfer funds from the current contract to the recipient contract.

As a result the `deposit` function invokes `safeTransferFrom` on a staked token to transfer the funds from the old contract to the new contract and assigns it to the EOA invoking the whole `restake` process.

The process described however always reverts, due to the fact the old contract (`TokenStakableVesting`, `StakingV2`) does not approve the new contract (`StakingV2`) to perform `safeTransferFrom` on its behalf.

Recommendations

- Consider increasing allowance during the restake process to match the amount which should be transferred and to make the function operational.

References

- https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#IERC20-approve-address-uint256-

Re-test

Implemented fix in `StakingV2.sol` approves transferring tokens from one `StakingV2` to another `StakingV2` instance during `deposit`.

Implemented fix in `TokenStakableVesting.sol` approves transferring tokens in the `restake` function.

3.2 Possibility to drain staked tokens through a malicious rewards vendor [High]

Severity	Status
High	Closed

Description

`StakingV2` allows users to stake tokens and receive rewards via multiple `StakingV2Vendor`. Each `StakingV2Vendor` is created per each ERC20 token which is allocated as a reward to users staking their tokens in parent (`StakingV2`) to `StakingV2Vendor` contract.

By creating a malicious reward matching the token that is staked in `StakingV2` contract it is possible to drain the staked tokens through the `StakingV2Vendor` which will treat staked tokens as reward tokens and transfer those to whoever claims it.

Technical details

In case a maintainer sets a reward token matching the base token that is staked in `StakingV2` contract, this action creates a `StakingV2Vendor` for staking token which starts allocating rewards to the participants of the staking protocol without enforcing enough allocation before the rewarding period starts. Typically either before the vendor starts rewarding users the reward tokens should be transferred to `StakingV2` contract instances. However if there is no such transfer and the rewarding token matches the staking token, the staked tokens will be incorrectly used as rewards.

As a result it is possible to imagine the following scenario:

- Alice, Bob, and Mallory staked 1.5mln `TPRO` tokens in `StakingV2` - 500k per person, each having 1/3rd of the rewards per block
- Mallory is able to convince someone with maintainer role to set a reward to 3mln `TPRO` tokens per block, starting immediately, but it is not necessary to immediately transfer those rewards
- Mallory in the next block claims rewards and as he has 1/3rd of the pool he gets 1mln in rewards. Later withdraws his 500k ending with 1.5mln `TPRO` when the contract is emptied as staked tokens of Alice and Bob were used as rewards and claimed by Mallory.

Due to the fact the `StakingV2Vendor` is created with allowance set to max and there is no distinction from `StakingV2Vendor's` perspective whether it transfers staked or reward tokens from his parent (`StakingV2`) contract it is possible to drain tokens using presented attack scenario.

As the functions necessary to perform this attack are either publicly available (`claim`) or guarded by a `onlyAuthority` modifier (`setTokenPerBlock`) it was not marked as a critical issue as it requires privileged role. Due to the fact the `onlyAuthority` contains owner and maintainer roles it was assumed that multiple wallets/contracts can pass this check, thus the finding was rated as high.

Recommendations

As rewarding the same token as a staked token seems to be a legitimate use-case it might be not possible to block such rewards completely, what would be the easiest solution to mitigate this issue. In such a case it is important to recognize which token is used as a reward and in case it is the same token as `parent.token()` the `StakingV2Vendor` should handle token claims differently. Claiming in such a case should differentiate between staked and reward tokens of the same type and prevent transferring staked (locked) tokens.

Re-test

Implemented fix correctly calculates available amount of rewards to send by subtracting staked amount stored in `tokenRealStaked` parent's property.

```
    function transferPendingRewards(uint256 pid, address to, uint256
amount) internal returns (uint256) {
        if (pid >= maxPid) {
            return 0;
        }
        if (amount == 0) {
            return 0;
        }
        uint256 tokenAmount = token.balanceOf(address(parent));

        // if reward token is the same as deposit token deduct its balance
from withdrawable amount
        if (tokenAmount != 0 && address(token) == address(parent.token()))
{
            for (uint i=0; i<maxPid && tokenAmount > 0; i++) {
                uint256 tokenRealStaked =
getParentPoolInfo(i).tokenRealStaked;
                tokenAmount = (tokenRealStaked >= tokenAmount) ? 0 :
tokenAmount.sub(tokenRealStaked);
            }
        }
        if (tokenAmount == 0) {
            return 0;
        }
        if (tokenAmount > amount) {
            tokenAmount = amount;
        }
        token.safeTransferFrom(address(parent), to, tokenAmount);
        return tokenAmount;
    }
```

3.3 Collected funds delivery to arbitrary address in SimplePresale [Medium]

Severity	Status
Medium	Accepted risk

Description

The `SimplePresale` allows to collect funds for teams starting in the blockchain ecosystem. The allocation is later used as a base to deliver the freshly minted tokens to the participants of the presale proportionally to their allocation.

Currently it is implemented as a two-step process which is done separately. First the presale has to be successful, so the owner of the presale can deliver collected funds to the team for which the presale was arranged. In the second step the team, based on the allocations from the `SimpePresale` contract, mints and delivers the freshly minted tokens.

As it is a two-step process, the final delivery part is crucial as if due to any human error funds are transferred to an inaccessible address, the funds might be stolen or lost forever.

Technical details

- SimplePresale.sol:

```
    function deliver(address addr, uint256 amount) external onlyOwner {
        // zero amount is allowed!
        require(isWithdrawAccepted(), 'Presale: unable to release yet!');

        uint256 unlockAmount = amount;
        uint256 actualAmount = token.balanceOf(address(this));
        if (unlockAmount > actualAmount) {
            unlockAmount = actualAmount;
        }
        require(unlockAmount > 0, 'Presale: funds were already
delivered!');

        token.safeTransfer(addr, unlockAmount);
        emit Delivered(msg.sender, address(token), unlockAmount);
    }
```

Recommendations

Consider using a claim pattern to deliver the collected amount of funds to a known accessible address for instance the `msg.sender` which is the owner and can be an EOA or multisignature wallet. Afterwards the funds can be transferred at will wherever it is needed.

In case it is not a desired method to allow owner transfer funds to his own account, even though currently it is also possible, consider implementing a basic access control and different roles: one to maange the presale itself and the second one allowed to execute delivery to its own address (`msg.sender`). This way it will be known upfront where the funds will be delivered and only the privileged address will be able to collect the funds to his own account.

Re-test

The team decided to keep the current functionality in its current form. It is a business requirement to be able to send the funds to any arbitrary address, thus it is not feasible to implement a claim pattern which allows a two-step process where the recipient can collect the funds by himself/herself.

The team accepts the risk that the funds can be sent to an inaccessible address and it is the responsibility of the owner of the presale to ensure it would not happen.

3.4 Duplicates allowed in StakingV2Balancer addStakingInstance [Medium]

Severity	Status
Medium	Closed

Description

Function `addStakingInstance` used in `StakingV2Balancer` allows duplicates resulting in an inconsistent number of rewards distributed to the `StakingV2` instances.

The function `addStakingInstance` is used to add allowed staking instances to the contract, so it is possible to distribute rewards to multiple instances at the same time. The function however accepts and stores the array of allowed instances allowing for duplicates.

As a result the `reward` function, which distributes the rewards to all staking instances calculates the same instances multiple times in the total number of tokens (to calculate the share), but effectively sets the `staking.setTokenPerBlock` only once with a fraction of the real tokens which are transferred to the staking instance. The end result is more tokens transferred to the staking instance than distributed through the rewards, resulting in internal state mismatch, which has to be corrected by withdrawing remaining tokens later in time after the staking/rewarding is over.

Technical details

- StakingV2Balancer.sol:

```
    function addStakingInstances(address[] memory stakingInstances,
uint256[] memory stakingLevels) public onlyOwner {
        require(stakingInstances.length > 0, 'StakingV2Balancer: staking
instances has to have at least one element!');
        address token = address(StakingV2(stakingInstances[0]).token());
        delete allowedStakingInstances;

        for (uint i=0; i<stakingInstances.length; ++i) {
            require(token ==
address(StakingV2(stakingInstances[i]).token()),
                'StakingV2Balancer: provided wrong staking!');
            allowedStakingInstances.push(
                StakingRoute({ stakingAddress:
address(stakingInstances[i]), stakingLevel: stakingLevels[i] })
            );
        }
        emit StakingInstanceChanged();
    }

    function rewardEven(IERC20 _token, uint256 _level, uint256 _amount,
uint256 _blockRange, uint256 _staked) public onlyOwner {
        uint256 tokenAmount;
        address addr;
```

```
        uint256 size;

        StakingV2 staking;
        for (uint i=0; i<allowedStakingInstances.length; i++) {
            if (allowedStakingInstances[i].stakingLevel >= _level) {
                size++;
            }
        }

        if (size == 0) return;

        for (uint i=0; i<allowedStakingInstances.length; i++) {
            if (allowedStakingInstances[i].stakingLevel >= _level) {
                addr = allowedStakingInstances[i].stakingAddress;
                staking = StakingV2(addr);

                tokenAmount = _amount / size;
                // @audit tokens transferred multiple times
                _token.safeTransferFrom(address(msg.sender), addr,
tokenAmount);
                if (_blockRange > 0) tokenAmount = tokenAmount /
_blockRange;
                staking.setTokenPerBlock(_token, tokenAmount, _blockRange);
// @audit reward overwritten
            }
        }
    }
```

Recommendations

Disallow duplicates in `allowedStakingInstances` as according to the development team comment it is not necessary to define same `StakingV2` instances with same or different staking levels in `StakingV2Balancer`.

Re-test

Implemented fix prevents duplicated staking instances.

- StakingV2Balancer.sol:

```
function addStakingInstances(address[] memory stakingInstances, uint256[]
memory stakingLevels) public onlyOwner {
…
   for (uint i=0; i<stakingInstances.length; ++i) {
            require(stakingAllowed[address(stakingInstances[i])] ==
false, 'StakingV2Balancer: duplicated instances!');
            require(token ==
address(StakingV2(stakingInstances[i]).token()),
                'StakingV2Balancer: provided wrong staking!');
            allowedStakingInstances.push(
                StakingRoute({ stakingAddress:
address(stakingInstances[i]), stakingLevel: stakingLevels[i] })
            );
            stakingAllowed[address(stakingInstances[i])] = true;
        }
…
```

3.5 Non standard delegate votes implementation [Medium]

Severity	Status
Medium	Accepted risk

Description

The token implements a subset of the current OpenZeppelin's `ERC20Votes`, but neither inherits from `ERC20Votes` nor implements it completely. From missing features currently there is no implementation of checkpoints which are used to track past voting power, which prevents double voting problems for on-going votes.

Moreover, the implementation relies on an external call to `delegate.moveSpendingPower` which if set to a malicious contract not implementing `moveSpendingPower` or implementing it in a malicious way can simply block the transfers completely or corrupt the internal state and voting power.

Additionally, from minor issues, the comments indicate the code is based on `ERC20Votes`, but even though the documentation indicates the event is emitted when tokens are moved there is no such event emission in the code.

Technical details

- Token.sol:

```
    function setDelegateAddress(ITokenDelegate _delegate) public onlyOwner
{
        require(address(_delegate) != address(0), 'Token: delegate address
needs to be different than zero!');
        delegate = _delegate;
        emit DelegateAddressChanged(address(delegate));
    }

    /**
     * @dev Move voting power when tokens are transferred.
     *
     * Emits a {DelegateVotesChanged} event.
     */
    function _afterTokenTransfer(address from, address to, uint256 amount)
    internal virtual override
    {
        super._afterTokenTransfer(from, to, amount);
        _moveSpendingPower(from, to, amount);
    }

    function _moveSpendingPower(address src, address dst, uint256 amount)
internal {
        if (src != dst && amount > 0 && address(delegate) != address(0)) {
            delegate.moveSpendingPower(src, dst, amount);
        }
```

```
    }
```

Recommendations

- Consider using a full standard implementation of `ERC20Votes` rather than own custom solution if there is an expected need in the future to implement voting,
- In case voting is not needed currently, but might be needed in the future, review OpenZeppelin's recommendation regarding tokens wrapping as described in references.

References

- https://docs.openzeppelin.com/contracts/4.x/governance#token
- https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#ERC20Votes
- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC20Votes.sol
- https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#ERC20Wrapper

Re-test

According to the team's statement: "*The mechanism is designed to send a signal to external contracts about funds being moved to preserve extensibility of the token without needing replacement or without needing convoluted proxying. It serves its purpose as it is.*"

It is worth mentioning that OpenZeppelins `ERC20Votes` implementation does not invoke an external call as it simply does a snapshot of the balance in the contract, so in case it is necessary to get the voting power at block X the `getPastVotes` method is invoked. This way there is no risk like in the current implementation when the owner is able to block all transfers by setting an incorrect or reverting delegate. In the reference OpenZeppelin's implementation, such case is not possible as checkpoints are stored locally.

3.6 Divide by zero in StakingV2Vendor updatePool [Low]

Severity	Status
Low	Partially Fixed

Description

Fuzz testing revealed an unhandled divide by zero exception in updatePool in `StakingV2Vendor` in `divisor = superPool.lastBlock.sub(pool.lastBlock);`.

Even though the error reverts the transaction, it was found the impact is minimal, thus the finding was marked as a low severity issue just indicating that insufficient validation is performed and additional unit tests should be implemented.

Technical details

- StakingV2Vendor.sol:

```
   function updatePool(uint256 pid) internal {
        if (pid >= maxPid) {
            return;
        }
        if (startBlock == 0 || block.number < startBlock) {
            return;
        }
        PoolInfo storage pool = poolInfo[pid];
        if (pool.lastBlock == 0) {
            pool.lastBlock = startBlock;
        }
        uint256 lastBlock = getLastRewardBlock();
        if (lastBlock <= pool.lastBlock) {
            return;
        }
        SuperPoolInfo memory superPool = getParentPoolInfo(pid);
        uint256 poolTokenRealStaked = superPool.tokenVirtStaked;
        if (poolTokenRealStaked == 0) {
            return;
        }

        uint256 multiplier = lastBlock.sub(pool.lastBlock);
        uint256 divisor = superPool.lastBlock.sub(pool.lastBlock);

        uint256 tokenRewarded =
superPool.tokenRewarded.sub(pool.tokenRewarded);
        uint256 tokenPerShare =
superPool.tokenPerShare.sub(pool.tokenPerShare);
        if (multiplier != divisor) {
            tokenRewarded = tokenRewarded.mul(multiplier).div(divisor);
            tokenPerShare = tokenPerShare.mul(multiplier).div(divisor);
        }
        pool.tokenRewarded = pool.tokenRewarded.add(tokenRewarded);
        pool.tokenPerShare = pool.tokenPerShare.add(tokenPerShare);
```

```
        pool.realTokenRewarded =
pool.realTokenRewarded.add(tokenRewarded.mul(tokenPerBlock).div(tokenParent
Precision));
        pool.realTokenPerShare =
pool.realTokenPerShare.add(tokenPerShare.mul(tokenPerBlock));
        pool.lastBlock = lastBlock;
    }
```

Steps to set the state where divide by zero is thrown.

Scenario 1:

- `setTokenPerBlock` to configure the reward
- forward a few blocks without invoking anything
- directly invoke `StakingV2Vendor.claim(0)` for the configured reward which will trigger `updatePool` in `StakingV2Vendor` leading to division by zero

Scenario 2:

- initialize `StakingV2`
- distribute rewards via balancer
- alice deposits funds - which results in updating the pool
- move a few blocks forward
- owner wants to reconfigure token rewards and invokes for already existing and working `StakingV2Vendor` and invoke `StakingV2.setTokenPerBlock(TPRO, <nr of blocks>)` which does not invoke parent `updatePool`
- divide by zero will be thrown

StakingV2.sol:

```
    function setTokenPerBlock(IERC20 _token, uint256 _tokenPerBlock)
public onlyAuthority {
        require(startBlock != 0, 'Staking: cannot add reward before setting
start block');
        require(address(_token) != address(0), 'Staking: token address
needs to be different than zero!');

        address addr = vendorInfo[address(_token)]; //@audit => read from
storage
        if (addr != address(0)) {
            IStakingV2Vendor vendor = IStakingV2Vendor(addr);
            uint256 _prevCloseBlock = vendor.closeBlock();
            if (_prevCloseBlock == 0 || block.number <= _prevCloseBlock) {
                _token.approve(address(vendor), MAX);
         // @audit directly invoked setTokenPerBlock without updatePool in
parent, leads to divide by zero in the inner function
                vendor.setTokenPerBlock(_tokenPerBlock,
vendor.startBlock(), vendor.closeBlock());
                return;
            }
        }
```

```
        setTokenPerBlock(_token, _tokenPerBlock, 0);
    }
```

Recommendations

Add additional unit tests to catch unhandled exceptions, add proper validation and keep the parent state updated when the underlying `StakingV2Vendor` rewards are modified.

Re-test

Function (`claim`) which leads to the divide by zero condition in scenario 1 was removed.

The `updatePool` method added in a loop in `setTokenPerBlock(IERC20 _token, uint256 _tokenPerBlock)` correctly synchronizes pools preventing divide by zero.

However it was discovered that changes were not added to the abstract class `IStakingV2Vendor` as well as `StakingV2Vendor` does not inherit from that class. Even though the usage of the class is correct in `StakingV2`, the methods in `IStakingV2Vendor` are inconsistent with the actual implementation in `StakingV2Vendor`. Due to the lack of the inheritance, there is no compilation error as currently `StakingV2Vendor` is missing a second `claim` implementation.

3.7 Insufficient documentation [Low]

Severity	Status
Low	Closed

Description

In general there is no proper documentation in the source code. In multiple places the comments were found to be generic and the majority of the code has no comments. There were instances where inline comments were provided, for instance explaining some backwards compatibility or the precision gadget parameter, however in general in more complex functions there was no such documentation either for the function itself or inline for specific parts of the code.

Proper documentation is necessary especially when there is no formal business documentation. Anyone who reads the source code, including security specialists, can learn from the proper inline or NatSpec comments to better understand the details of the implementation or the initial requirements.

Technical details

- Almost all functions are missing proper comments,
- In some cases generic comments are used, e.g. restake in `TokenStakableVesting` could explain the process more thoroughly in the comment:

```
    /**
     * @dev Sends specific amount of released tokens and send it to sender
     */
    function restake(uint256 pid, address addr, uint256 saving, uint256
timerange) public {
        restake(pid, addr, saving, currentBalance(msg.sender), timerange);
    }
```

There are only a few instances of well-written inline documentation:

```
…
tokenPerBlock = 1e4; // in this interface tokenPerBlock serves purpose as a
precision gadget
…
// start block has to remain same regardless of current timestamp and block
range
            _startBlock = IStakingV2Vendor(addr).startBlock();
…
```

Recommendations

Proper documentation should describe the goal of the function, the parameters and the results as a bare minimum, however it should not be limited only to this generic information. Well-written documentation should include specification of the function, mechanisms which are implemented, explained calculations as well as inline comments in important parts.

References

- https://docs.soliditylang.org/en/v0.8.13/natspec-format.html
- https://github.com/nascentxyz/simple-security-toolkit/blob/main/audit-readiness-checklist.md
- https://github.com/nascentxyz/simple-security-toolkit/blob/main/development-process.md

Re-test

The Team updated inline comments in a few more complex functions.

3.8 Lack of zero check for divisor in StakingV2Balancer [Low]

Severity	Status
Low	Closed

Description

The function `rewardProp` in `StakingV2Balancer` performs division, but does not verify whether the divisor (`_staked`) is greater than zero, which can lead to unhandled division by zero.

As the contract is a utility that can be used in complex rewards distribution and usually the plain `reward` function will be used, which checks whether `_staked` is greater than zero, the finding was marked as low. However, due to the fact the `rewardProp` function as well as the second `rewardEven` both are public functions the input parameters should be validated properly or the visibility should be reduced, so the owner is able to invoke those functions only through the `reward` method.

Technical details

```
function rewardProp(IERC20 _token, uint256 _level, uint256 _amount, uint256
_blockRange, uint256 _staked) public onlyOwner {
        uint256 tokenAmount;
        address addr;

        StakingV2 staking;
        for (uint i=0; i<allowedStakingInstances.length; i++) {
            if (allowedStakingInstances[i].stakingLevel >= _level) {
                addr = allowedStakingInstances[i].stakingAddress;
                staking = StakingV2(addr);
                ( ,, uint256 tokenRealStaked,,,,, ) = staking.poolInfo(0);
                if (tokenRealStaked > 0) {
                    tokenAmount = _amount * tokenRealStaked / _staked;
                    _token.safeTransferFrom(address(msg.sender), addr,
tokenAmount);
                    if (_blockRange > 0) tokenAmount = tokenAmount /
_blockRange;
                    staking.setTokenPerBlock(_token, tokenAmount,
_blockRange);
                }
            }
        }
    }
```

Recommendations

- Restrict visibility of `rewardEven` and `rewardProp` functions or implement proper parameters validation if those functions are supposed to be exposed.

Re-test

Implemented argument check prevents divide by zero.

```
function rewardProp(IERC20 _token, uint256 _level, uint256 _amount, uint256
_blockRange, uint256 _staked) public onlyOwner {
        require(_staked > 0, 'StakingV2Balancer: divisor needs to be higher
than zero!');
…
```

3.9 Missing 2-step ownership transfer [Low]

Severity	Status
Low	Accepted risk

Description

The contracts implement a basic `Ownable` modifier from OpenZeppelin. The implementation does not verify whether the new owner has access to the wallet, thus in case of a mistake during the ownership transfer the contract might be transferred to an inaccessible address.

The ownership transfer should be a two-steps process, whether during the first step the new owner is set and in the second step to finish the transfer, the new owner has to claim the ownership to prove access to the wallet.

Technical details

Affected contracts which should consider using a two-step ownership transfer process:

- `SimplePresale.sol`
- `StakingV2.sol`
- `StakingV2Balancer.sol`
- `Token.sol`
- `TokenVesting.sol`
- `TokenStakableVesting.sol`

Recommendations

- Consider implementing a two-step ownership transfer process to prevent losing access to the contract in case the ownership is transferred to an inaccessible address.
- Consider using a multi-signature wallet to own the contract.

Re-test

The Team decided to not implement 2-step ownership transfer. The team prefers transferring ownership without the confirmation. No changes were implemented.

3.10 Mixed timestamp and timespan variables in deposit function in StakingV2 [Low]

Severity	Status
Low	Partially fixed

Description

`StakingV2` allows users to deposit funds which can be locked for a given `timerange` when staking has started. However the `require` statement is validating the `timerange` parameter and compares it to a pool's parameter which suggests it is a timestamp rather than a timespan. `timerange` parameter is a timespan for which staked funds should be locked.

The information from the development team states the `pool.lockupTimestart` is a timespan, however the analysis of the source code and places where it is used suggests it is incorrectly named and it is a timespan. In fact `pool.lockupTimestart` is `_minPoolTimer` parameter from `StakingV2` constructor.

Technical details

- StakingV2.sol

```
function _deposit(uint256 pid, address from, address addr, uint256 amount,
uint256 timerange) internal {

…
require(timerange <= pool.lockupTimerange && timerange >=
pool.lockupTimestart,
            'Staking: cannot lock funds for that amount of time!');
…
user.lockedTimestamp = block.timestamp + timerange;
…
}

    constructor(IERC20 _token, uint256 _minPoolTimer, uint256
_maxPoolTimer, uint256 _minAmount, uint256 _maxAmount, uint256 _poolLimit)
{
        require(address(_token) != address(0), 'Staking: token address
needs to be different than zero!');
        token = _token;          // @audit token for staking
        minAmount = _minAmount; // @audit min amount for staking token
        maxAmount = _maxAmount; // @audit max amount for staking token
        addPool(_minPoolTimer, _maxPoolTimer, _poolLimit);
        tokenPerBlock = 1e4; // in this interface tokenPerBlock serves
purpose as a precision gadget

        _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
        _setRoleAdmin(MAINTAINER_ROLE, ADMIN_ROLE);

        _setupRole(ADMIN_ROLE, address(this));
```

```
        _setupRole(MAINTAINER_ROLE, owner());
    }

    function addPool(uint256 _lockupTimestart, uint256 _lockupTimerange,
uint256 _tokenTotalLimit) internal {
        require(maxPid < 10, 'Staking: Cannot add more than 10 pools!');

        poolInfo.push(PoolInfo({
            lastBlock: 0,
            tokenPerShare: 0,
            tokenRealStaked: 0,
            tokenVirtStaked: 0,
            tokenRewarded: 0,
            tokenTotalLimit: _tokenTotalLimit,
            lockupTimerange: _lockupTimerange,
            lockupTimestart: _lockupTimestart
        }));
        maxPid++;

        emit PoolAdded(_lockupTimestart, _lockupTimerange,
_tokenTotalLimit);
    }
```

Recommendations

- Make sure the names of the variables and properties reflect their type.

Re-test

The variable names were renamed to reflect min and max time range which is allowed while depositing funds and an additional check was added to check the maximum allowed time range.

- StakingV2.sol

```
function _deposit(uint256 pid, address from, address addr, uint256 amount,
uint256 timerange) internal {
…
require(timerange <= pool.lockupMaxTimerange && timerange >=
pool.lockupMinTimerange,
            'Staking: cannot lock funds for that amount of time!');

…
```

As currently both values can be set in the constructor which lacks verification (as well as `addPool`) whether `lockupMaxTimerange >= lockupMinTimerange`, it is possible to initialize the contract with values which effectively render the `deposit` function non-operational when min time range is bigger than max time range.

- StakingV2.sol

```
    constructor(IERC20 _token, uint256 _minPoolTimer, uint256
_maxPoolTimer, uint256 _minAmount, uint256 _maxAmount, uint256 _poolLimit)
{
```

```
        require(address(_token) != address(0), 'Staking: token address
needs to be different than zero!');
        token = _token;
        minAmount = _minAmount;
        maxAmount = _maxAmount;
        addPool(_minPoolTimer, _maxPoolTimer, _poolLimit);
        tokenPerBlock = 1e4; // in this interface tokenPerBlock serves
purpose as a precision gadget

        _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
        _setRoleAdmin(MAINTAINER_ROLE, ADMIN_ROLE);

        _setupRole(ADMIN_ROLE, address(this));
        _setupRole(MAINTAINER_ROLE, owner());
    }

    function addPool(uint256 _lockupMinTimerange, uint256
_lockupMaxTimerange, uint256 _tokenTotalLimit) internal {
        require(maxPid < 10, 'Staking: Cannot add more than 10 pools!');

        poolInfo.push(PoolInfo({
            lastBlock: 0,
            tokenPerShare: 0,
            tokenRealStaked: 0,
            tokenVirtStaked: 0,
            tokenRewarded: 0,
            tokenTotalLimit: _tokenTotalLimit,
            lockupMaxTimerange: _lockupMaxTimerange,
            lockupMinTimerange: _lockupMinTimerange
        }));
        maxPid++;

        emit PoolAdded(_lockupMinTimerange, _lockupMaxTimerange,
_tokenTotalLimit);
    }
```

It is recommended to add missing validation to prevent deployment of a non-operational contract.

3.11 Missing important address in emit event while adding staking to allowed list [Low]

Severity	Status
Low	Closed

Description

`StakingV2` and similar contracts like `TokenStakableVesting`, `StakingV2Balancer` are missing address of staking instance in emitted event for each modification of `allowedStakingInstances`.

`allowedStakingInstances` can be modified to add or remove addresses which can be used in `restake` functionality. The function `addStakingInstances` however emits only a generic event that the property of the contract was modified, but it does not emit the exact address. As most of the off-chain tools, especially for end-users do not automatically decode the data of the event it makes it harder to track the changes of important properties of the contract.

Technical details

```
    function addStakingInstances(address[] memory stakingInstances, bool
status) public onlyOwner {
        for (uint i=0; i<stakingInstances.length; ++i) {
            allowedStakingInstances[stakingInstances[i]] = status;
        }
        emit StakingInstanceChanged();
    }
```

Affected contracts:

- `TokenStakableVesting`
- `StakingV2`
- `StakingV2Balancer`

Recommendations

Consider emitting an event which contains the address of the allowed/disallowed staking address for easier off-chain monitoring.

Re-test

The Team decided the additional data in events is unnecessary and they do not want to spend more gas when a list of allowed staking instances is updated. Off-chain tools need to decode data manually. No changes were implemented.

3.12 Multiple tests failing [Low]

Severity	Status
Low	Closed

Description

Multiple tests were found to be failing, mostly in `SimplePresale.sol` after launching according to the provided documentation. Proper CI configuration should prevent such code from being committed as it is bad practice to have code which does not pass unit tests created by the development team.

During a security assessment unit tests are mostly helpful to understand some complex use-cases which were covered by the development team or to check how well the codebase is tested in general, but the exact reason why each of the tests was failing was out of the scope of the assessment. A few test cases were coded again in Foundry to verify whether presale works as intended in case of automatically closed presale if a goal is reached or when it is not allowed to do additional allocation after the sale is closed and those tests did not fail, meaning there might be something broken either with the tests configuration or tests initialization. The exact reason was not pursued.

Technical details

Truncated and cleaned test runner listing:

```
$ npm run test

14 failing

  1) Contract: SimplePresale
       when sale has been opened for whitelisted
           when allocating funds
             rejects if allocation target has been reached:
     AssertionError: Expected an exception but none was received

  2) Contract: SimplePresale
       when sale has been opened for whitelisted
             sale automatically closes if enough funds were acquired:

      AssertionError: expected false to equal true
      + expected - actual

  3) Contract: SimplePresale
       when sale has been opened for whitelisted
             sale automatically closes if amount of funds were in range of
margin:

      AssertionError: expected false to equal true
      + expected - actual

  4) Contract: SimplePresale
```

```
       when sale has been opened for everyone
           when allocating funds
             rejects if allocation target has been everyone:
     AssertionError: Expected an exception but none was received


  5) Contract: SimplePresale
       when sale has been opened for everyone
             is possible to close sale if enough funds were acquired:

      AssertionError: expected false to equal true
      + expected - actual

  6) Contract: SimplePresale
       when sale has been opened for everyone
             is possible to close sale if amount of funds were in range of
margin:

      AssertionError: expected false to equal true
      + expected - actual

  7) Contract: SimplePresale
       when sale has been closed
             defines its status as not active:

      AssertionError: expected true to equal false
      + expected - actual

  8) Contract: SimplePresale
       when sale has been closed
             defines participation status as not active:

      AssertionError: expected true to equal false
      + expected - actual

  9) Contract: SimplePresale
       when sale has been closed
             defines its status as closed:

      AssertionError: expected false to equal true
      + expected - actual

  10) Contract: TokenStaking
       when staking is configured for single-asset
             rejects if deposit is made without approval:

      Wrong kind of exception received
      + expected - actual
```

Recommendations

- Consider adding GitHub actions to configure CI/CD to verify no tests are failing when the code is modified.
- Consider adding slither to the CI/CD pipeline to catch the common bugs.

Re-test

The tests do work on development team's machines and in the end it was identified that some specific requirements to launch the tests were missing in the README. First of all the older version of Ganache (before 7.x) is needed as the development team suggested there were some non deterministic testing issues after the London hard fork. Secondly, the `package.json` file specifies the `"@openzeppelin/contracts": "^4.2.0"` version which automatically fetched the newest dependency, by the time of writing this report it is 4.6.0. The newest OpenZeppelin contracts library has breaking changes in allowance error handling which caused unit tests to fail as different revert messages were expected.

It is recommended to either adjust the tests to comply with the newest version of the library from OpenZeppelin or lock the version to 4.2.0 only. Still it is recommended to create a docker or other similar deployment strategy encapsulating all dependencies and settings that can be used in CI/CD pipelines.

As it was possible to launch all tests, the finding no longer applies and was marked as closed.

3.13 Never used variable - maxAmount [Low]

Severity	Status
Low	Closed

Description

`StakingV2` in constructor accepts the `_maxAmount` parameter, assigns it to the property stored in the storage as `maxAmount`, but later it was not found to be used anywhere in the source code.

The property seems to be not used, but occupies the storage slot wasting gas during the contract's creation.

Technical details

- StakingV2.sol:

```
    constructor(IERC20 _token, uint256 _minPoolTimer, uint256
_maxPoolTimer, uint256 _minAmount, uint256 _maxAmount, uint256 _poolLimit)
{
        require(address(_token) != address(0), 'Staking: token address
needs to be different than zero!');
        token = _token;          // @audit token for sta
        minAmount = _minAmount; // @audit min amount for staking token
        maxAmount = _maxAmount; // @audit max amount for staking token
        addPool(_minPoolTimer, _maxPoolTimer, _poolLimit);
        tokenPerBlock = 1e4; // in this interface tokenPerBlock serves
purpose as a precision gadget

        _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
        _setRoleAdmin(MAINTAINER_ROLE, ADMIN_ROLE);

        _setupRole(ADMIN_ROLE, address(this));
        _setupRole(MAINTAINER_ROLE, owner());
    }
```

Recommendations

Remove unused properties, both to keep the code clean and do not waste contract's storage.

Re-test

The team left the property and added a check as the contract was missing `maxAmount` validation in deposit. The following code was added and verified whether the user's deposit has not reached the maximum value.

- StakingV2.sol:

```
   require(maxAmount == 0 || user.amount + amount <= maxAmount, 'Staking:
amount needs to be lesser');
```

3.14 No full unit test coverage [Low]

Severity	Status
Low	Closed

Description

The unit tests do not provide full coverage of the smart contracts. Testing especially in smart contracts is crucial to provide sufficient confidence all business cases are covered and all functionality is operational.

For instance it was found that `TokenVesting` and `TokenStakableVesting` tests are identical, besides the name of the contract which is initialized, despite the fact the latter implements additional restake functionality. Additionally it was found that this additional functionality was found to be non-operational, what should be covered and discovered through proper unit testing.

Recommendations

- Implement unit tests,
- Implement business logic tests,
- Implement fuzzing tests,
- Consider using a coverage tool to verify which parts of the code were not covered by unit tests.

References

- https://ethereum.org/en/developers/tutorials/solidity-and-truffle-continuous-integration-setup/#step-1-create-a-metacoin-project-and-install-coverage-tools

Re-test

Missing tests for `TokenStakableVesting` were added, thus the case which was missing previously is currently correctly tested.

Coverage tools as well as CI/CD pipelines are still not configured, thus it is recommended to review linked references and consider using e.g. GitHub Actions to create a fully automated CI/CD pipeline.

3.15 No support for fee-on-charge tokens as rewards [Low]

Severity	Status
Low	Accepted risk

Description

`StakingV2` allows for any ERC20 token as a reward, but does not implement any check for deflationary tokens. Same is true for `StakingV2Balancer`. Some examples of deflationary tokens are tokens which can charge a fee for transactions, e.g. fee-on-transfer. As a result when a transfer is invoked with a given amount the fee is calculated and deducted from the initial transferred amount. In the end less tokens are received than anticipated.

Even though the staking token, according to collected information, is not such a deflationary token, the rewards can be any ERC20 tokens, which includes deflationary tokens with fee-on-charge. Taking this into account the implementation should consider deflationary tokens as it can lead to inconsistencies in rewards allocated for users and rewards stored in the contract or prevent using such types of tokens.

The most affected contract is `StakingV2Balancer` as the contract does not check whether the reward token is a deflationary token. The contract takes the amount to distribute and in case of a deflationary token the amount distributed via `reward` method will be greater than the sum of the tokens actually transferred to the staking instances. The same time the reward configuration for each `StakingV2` instance set via `setTokenPerBlock` will use not the actually transferred amount of tokens, but the initial amount what leads to inconsistency and greater reward calculated by `StakingV2Vendor` compared to the number of tokens transferred to the `StakingV2` instance as rewards.

For `StakingV2` it is less important as the usage of such tokens make it mandatory for the person configuring the rewards to directly transfer enough tokens, so amount + token's fee-on-transfer fee to match the amount set in rewards configuration. Still the end users claiming the reward will receive less tokens than expected as it is also not verified, but it is a characteristic of such a reward token with fee-on-transfer capabilities, thus should be considered as intended behavior.

Technical details

- StakingV2Balancer:

```
    function rewardEven(IERC20 _token, uint256 _level, uint256 _amount,
uint256 _blockRange, uint256 _staked) public onlyOwner {
        uint256 tokenAmount;
        address addr;
        uint256 size;

        StakingV2 staking;
```

```
        for (uint i=0; i<allowedStakingInstances.length; i++) {
            if (allowedStakingInstances[i].stakingLevel >= _level) {
                size++;
            }
        }

        if (size == 0) return;

        for (uint i=0; i<allowedStakingInstances.length; i++) {
            if (allowedStakingInstances[i].stakingLevel >= _level) {
                addr = allowedStakingInstances[i].stakingAddress;
                staking = StakingV2(addr);

                tokenAmount = _amount / size;
                _token.safeTransferFrom(address(msg.sender), addr,
tokenAmount);
                if (_blockRange > 0) tokenAmount = tokenAmount /
_blockRange;
                staking.setTokenPerBlock(_token, tokenAmount, _blockRange);
            }
        }
    }
```

Recommendations

If deflationary fee-on-transfer tokens should not be supported, consider adding in `StakingV2Balancer` an invariant checking the balance of the recipient `StakingV2` instance before the transfer and after the transfer. If both balances are the same, the token did not charge a fee on the transaction itself.

In case of `StakingV2` and in general ERC20 tokens which can charge such transfer fees, do not include such an invariant as it might break the logic of the claim functionality. Due to the fact the reward configuration and token transfer are two separate actions such an invariant could lock deflationary rewards tokens in the `StakingV2` contract as such tokens would be accepted to be rewards, but could never be claimed by end users, due to invariant check. Keep in mind though, that configuring rewards and reimbursing the contract with deflationary tokens should include internal fees to be taken into consideration.

Re-test

The team has no intention to support fee-on-transfer tokens or to block such tokens. Even an invariant check does not necessarily protect completely from such tokens, thus it was decided such safety mechanisms are not necessary and will not be implemented. This requirement is outside of the business scope.

3.16 Unused function parameter in StakingV2Balancer [Low]

Severity	Status
Low	Closed

Description

The parameter of the `rewardEven` function in `StakingV2Balancer` was found to be not used.

The parameter seems to be unnecessary and looks like a leftover from the other similar function which uses it.

Technical details

```
function rewardEven(IERC20 _token, uint256 _level, uint256 _amount, uint256
_blockRange, uint256 _staked) public onlyOwner {
        uint256 tokenAmount;
        address addr;
        uint256 size;

        StakingV2 staking;
        for (uint i=0; i<allowedStakingInstances.length; i++) {
            if (allowedStakingInstances[i].stakingLevel >= _level) {
                size++;
            }
        }

        if (size == 0) return;

        for (uint i=0; i<allowedStakingInstances.length; i++) {
            if (allowedStakingInstances[i].stakingLevel >= _level) {
                addr = allowedStakingInstances[i].stakingAddress;
                staking = StakingV2(addr);

                tokenAmount = _amount / size;
                _token.safeTransferFrom(address(msg.sender), addr,
tokenAmount);
                if (_blockRange > 0) tokenAmount = tokenAmount /
_blockRange;
                staking.setTokenPerBlock(_token, tokenAmount, _blockRange);
            }
        }
    }
```

Recommendations

- Remove unnecessary parameters.

Re-test

Parameter was removed.

3.17 Gas optimization - external functions instead of public [Info]

Severity	Status
Info	Closed

Description

Functions which are not invoked in other methods and are exposed publicly should be declared with an external modifier to save gas.

Technical details

Example list of functions from nearly all contracts which can be declared external:

```
addStakingInstances(address[],bool) should be declared external:
      - TokenStakableVesting.addStakingInstances(address[],bool)
(Token/TokenStakableVesting.sol#23-29)

setTokenAddress(IERC20) should be declared external:
      - TokenVesting.setTokenAddress(IERC20)
(Token/TokenVesting.sol#49-54)

addStakingInstances(address[],uint256[]) should be declared external:
      - StakingV2Balancer.addStakingInstances(address[],uint256[])
(Staking/StakingV2Balancer.sol#24-37)
reward(IERC20,uint256,uint256,uint256) should be declared external:

initAntibot(uint256,uint256,uint256) should be declared external:
      - AntiBot.initAntibot(uint256,uint256,uint256)
(Token/AntiBot.sol#27-35)

setFactoryAddress(IStakingV2Factory) should be declared external:
      - StakingV2.setFactoryAddress(IStakingV2Factory)
(Staking/StakingV2.sol#132-136)

…<TRUNCATED>...
```

A full list can be obtained by scanning the contracts using slither.

Recommendations

- Review listed functions and declare them external to save gas,
- Add slither to the CI/CD pipeline to catch common issues.

References

- https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external

Re-test

Team decided it is unnecessary optimization. No changes were implemented.

3.18 No emit event in StakingV2Balancer reward [Info]

Severity	Status
Info	Closed

Description

The `reward` and two similar related functions (`rewardEven`, `rewardProp`) are missing emit events indicating the reward was distributed to the `StakingV2` instances.

Events are useful to easily monitor stage changes in off-chain tools, thus for all logical actions which change the state it is a good practice to emit relevant events. Even though the underlying `StakingV2` and corresponding `StakingV2Vendor` instances do emit reward events when the `setTokenPerBlock` is invoked for off-chain analysis it might be beneficial to easily monitor actions done via the `StakingV2Balancer` as well.

Recommendations

Consider adding an emit event for reward distribution via `StakingingV2Balancer`.

Re-test

Team decided it is not necessary to emit events. No changes were implemented.