Competitive Security Assessment

Tonka_Finance_Staking_Yield

Jan 15th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

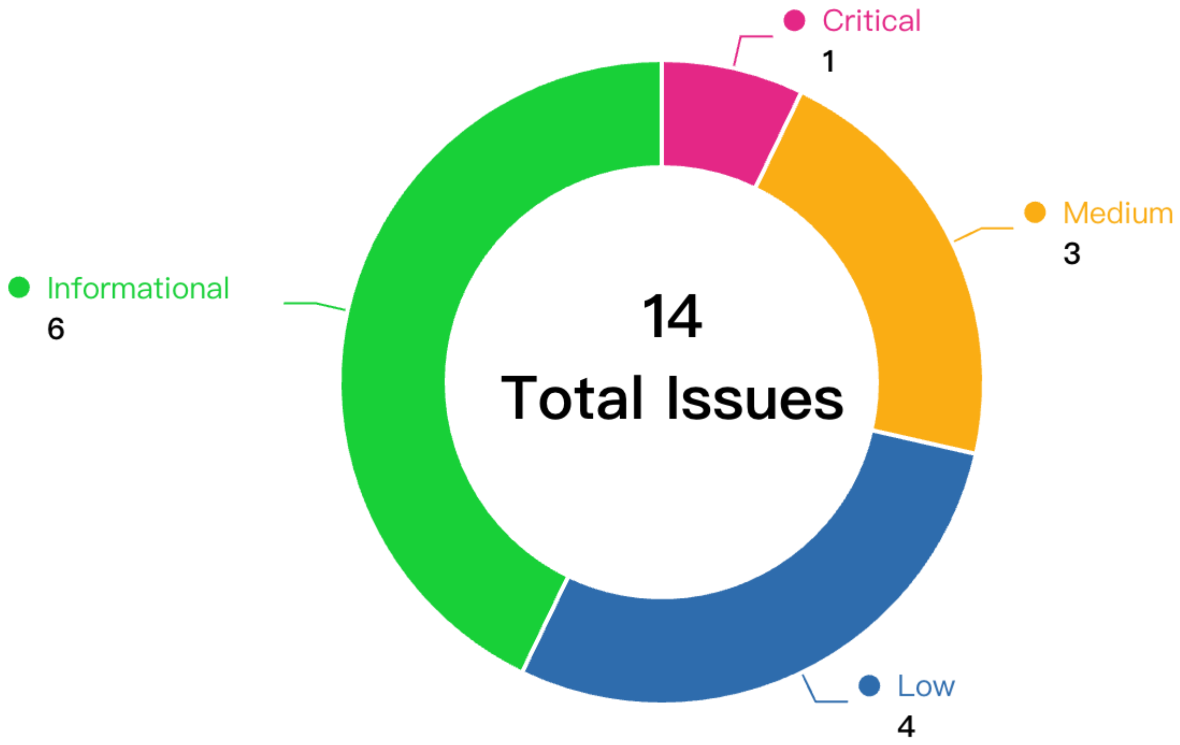
Project Detail

Project Name	Tonka_Finance_Staking_Yield
Platform & Language	Solidity
Codebase	• https://github.com/Tonka-Finance/Tonka-Contracts • audit commit - 728048d72d1a35fcba4d6bd3667e4f98839c28ed • final commit - b378b379dbeb7c3fc5a3035e69ea454a86d375ab
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
contracts/staking/DoubleStaking.sol	**f6009f01608c69070a5d10cb1e7d348d7e47ca777cebf4addf0f2011a8c0408a**
contracts/staking/SingleStaking.sol	**e8a7a1d585a4225d47a69b2aea718c643b31919e39afa5216f8204dac65c6528**
contracts/token/TokaVesting.sol	**de2978d7dbcbc8fa600d0a3a963ccf700d0782f7d438545510f7494aa016543c**
contracts/token/EsTokaToken.sol	**dd5c4b49b394457c545ab45bcaf225c1f0f9d124da7d6b0619f867cc9a93708e**
contracts/token/TokaToken.sol	**ee03b853490625eef561fb508a270ddc27b1e5accca77b954ae1a2d3262f7616**

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
TFS-1	Improper handling of lockedAmount may result in vesting being unable to be executed	Logical	Critical	Fixed	8olidity
TFS-2	Potential reentrancy attack to steal rewards when `tokenA` is set to an ERC777 token	Logical	Medium	Fixed	Hacker007
TFS-3	Staking is only compatible with tokens of decimal 1e18	Logical	Medium	Fixed	Hacker007, 0xffchain

TFS-4	**Renounce ownership should be disabled**	**Logical**	**Medium**	**Fixed**	**crjr0629**
TFS-5	`call()` **should be used instead of** `transfer()` **on an address payable**	**Language Specific**	**Low**	**Fixed**	**8olidity, 0xffchain, Hacker007**
TFS-6	**Missing calls to** `__Pausable_init()` **function in** `TokaVesting`	**Logical**	**Low**	**Fixed**	**Hacker007**
TFS-7	**Unprotected initializer**	**Logical**	**Low**	**Fixed**	**Hacker007**
TFS-8	**Wrong lastStakeTime update logic in** `SingleStaking::stake()` **and** `DoubleStaking ::stake()` **function**	**Logical**	**Low**	**Fixed**	**Hupixiong3**
TFS-9	**Event Vesting should be emitted with vesting amount instead of claimed amount**	**Code Style**	**Informational**	**Fixed**	**minhquanym**
TFS-10	**Claim event should be emitted in function** `vesting()`	**Code Style**	**Informational**	**Fixed**	**minhquanym**
TFS-11	**Supply cap is not enforced in** `TokaToken`	**Logical**	**Informational**	**Fixed**	**minhquanym**
TFS-12	**Missing Zero Address Check**	**Code Style**	**Informational**	**Fixed**	**8olidity, helookslikeme, LiRiu**
TFS-13	**redundant variables** `cap`	**Code Style**	**Informational**	**Fixed**	**8olidity, Hupixiong3, Hacker007, crjr0629**
TFS-14	**Unified optimization of the coinage process in** `TokaVesting::claim()`	**Code Style**	**Informational**	**Fixed**	**Hupixiong3**

TFS-1:Improper handling of lockedAmount may result in vesting being unable to be executed

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	8olidity

Code Reference

- code/contracts/token/TokaVesting.sol#L66
- code/contracts/token/EsTokaToken.sol#L91-L95
- code/contracts/staking/SingleStaking.sol#L95-L106

```
66:uint256 unlockedEsToken = esTokaToken.balanceOf(msgSender) - esTokaToken.lockedAmount(msgSender);

91:function lock(address _to, uint256 _amount) external onlyWhiteLister {
92:        uint256 unlocked = balanceOf(_to) - lockedAmount[_to];
93:        require(_amount <= unlocked, "insufficient unlocked balance");
94:        lockedAmount[_to] += _amount;
95:    }

95:function stake(uint256 _amount) external {
96:        address msgSender = msg.sender;
97:        UserInfo storage user = userInfo[msgSender];
98:
99:        updateReward();
100:
101:        // If the user had staked before, harvest the reward first
102:        if ((user.amount + user.loyalAmount) > 0) {
103:            uint256 pending = ((user.amount + user.loyalAmount) * accRewardPerShare) / 1e18 - user.rewardDebt;
104:
105:            if (pending > 0) {
106:                IMintableToken(esTokaToken).mint(msgSender, pending);
```

Description

8olidity : In the `vesting` function, the `burn` function of `esTokaToken` is called, but it doesn't handle the `lockedAmount` value. Let's take a look at an example:

1. Alice calls `SingleStaking::stake()`. Let's say Alice mints 2 `esTokaToken` at this point.

```
function stake(uint256 _amount) external {
    address msgSender = msg.sender;
    UserInfo storage user = userInfo[msgSender];

    updateReward();

    // If the user had staked before, harvest the reward first
    if ((user.amount + user.loyalAmount) > 0) {
        uint256 pending = ((user.amount + user.loyalAmount) * accRewardPerShare) / 1e18 - user.rewar
dDebt;

        if (pending > 0) {
            IMintableToken(esTokaToken).mint(msgSender, pending);
            emit Harvest(msgSender, pending);
        }
    }
}
```

2. Alice locks 2 `esTokaToken`:

```
function lock(address _to, uint256 _amount) external onlyWhiteLister {
    uint256 unlocked = balanceOf(_to) - lockedAmount[_to];
    require(_amount <= unlocked, "insufficient unlocked balance");
    lockedAmount[_to] += _amount;
}
```

3. Alice then calls `TokaVesting::vesting()`, which burns Alice's `esTokaToken`:

```
esTokaToken.burn(msgSender, _amount);
```

But the burn function does not process the value of lockedAmount, and lockedAmount is still the previous value.

4. Afterward, if Alice uses `vesting(amount)` with an amount smaller than the previous value, it will not execute correctly.

```
uint256 unlockedEsToken = esTokaToken.balanceOf(msgSender) - esTokaToken.lockedAmount(msgSender);
require(_amount <= unlockedEsToken, "insufficient unlocked balance");
```

Recommendation

8olidity : It is recommended to handle the value of lockedAmount when burning the function

Client Response

Fixed,add a handle for lockedAmount in burn func commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/fe1d01945cdf3efb08873ec7ff8ed4cfd2132183

TFS-2:Potential reentrancy attack to steal rewards when `tokenA` is set to an ERC777 token

Category	Severity	Client Response	Contributor
Logical	**Medium**	**Fixed**	**Hacker007**

Code Reference

- code/contracts/staking/DoubleStaking.sol#L180-L211
- code/contracts/staking/DoubleStaking.sol#L277-L292

```
function withdraw(uint256 _poolId, uint256 _amountA) external {
        address msgSender = msg.sender;
        PoolInfo storage pool = poolInfo[_poolId];
        UserInfo storage user = userInfo[_poolId][msgSender];

        require(user.amountA >= _amountA, "Insufficient balance");
        require(user.lastStakeTime + withdrawalCooldown <= block.timestamp, "withdrawal cooldown");

        updateReward(_poolId);

        uint256 pending = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18 - user.rewardDebt;
        if (pending > 0) {
            IMintableToken(esTokaToken).mint(msgSender, pending);
            emit Harvest(msgSender, pending);
        }

        // Fixed ratio of tokenA to tokenB
        uint256 amountB = (_amountA * pool.stakeRatio) / 1000;

        user.amountA -= _amountA;
        user.amountB -= amountB;

        pool.supplyA -= _amountA;
        pool.supplyB -= amountB;

        // Transfer tokens to user
        IERC20(pool.tokenA).safeTransfer(msgSender, _amountA);
        IERC20(pool.tokenB).safeTransfer(msgSender, amountB);

        user.rewardDebt = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18;
        emit Withdraw(msgSender, _amountA, amountB);
    }

uint256 pending = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18 - user.rewardDebt;
        if (pending > 0) {
            IMintableToken(esTokaToken).mint(msgSender, pending);
            emit Harvest(msgSender, pending);
        }
        
        uint256 loyalAmountB = (_amountA * pool.stakeRatio) / 1000;
        
        user.loyalAmountA -= _amountA;
        user.loyalAmountB -= loyalAmountB;
        
        pool.loyalSupplyA -= _amountA;
        pool.loyalSupplyB -= loyalAmountB;
        
        IERC20(pool.tokenA).safeTransfer(msgSender, _amountA);
        IERC20(pool.tokenB).safeTransfer(msgSender, loyalAmountB);
```

Description

Hacker007 : Per the EIP-777, ERC777 tokens are backward-compatible with ERC20 and can be used to set as `tokenA` for a pool. If an ERC777 token is set as tokenA, a reentrancy attack may allow the malicious user to a bunch of reward tokens when calling the function `withdraw()`.

```
    function withdraw(uint256 _poolId, uint256 _amountA) external {
//...
        updateReward(_poolId);

        uint256 pending = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18 - user.rewardDebt;
        if (pending > 0) {
            IMintableToken(esTokaToken).mint(msgSender, pending);
            emit Harvest(msgSender, pending);
        }
//...
        // Transfer tokens to user
        IERC20(pool.tokenA).safeTransfer(msgSender, _amountA);
        IERC20(pool.tokenB).safeTransfer(msgSender, amountB);

        user.rewardDebt = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18;
        emit Withdraw(msgSender, _amountA, amountB);
    }
```

Consider this exploit scenario:

1. The msgSender calls the `withdraw` function with 1 wei amount, and a pending reward token is minted to `msgSender`.
2. After the `msgSender` receives `_amountA` tokens, the receiver's hook will be called.

3. The hook calls the `withdraw` function again, this time `user.rewardDebt` is updated, and a pending reward token can still be minted to `msgSender`.
4. After the `msgSender` receives `_amountA` tokens again, the receiver's hook will be called. An attacker can do steps 1- 3 many times before running out of gas, and he gains a giant amount of reward tokens.

The same issue happens in `loyalWithdraw()`.

Recommendation

Hacker007 : Add reentrancy guards guard from openzeppelin to the aforementioned functions.

Client Response

Fixed.fix: add ReentrancyGuard for withdraw and loyalWithdraw commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/a597c857c55527dc53a397e09c8381e095965c70

TFS-3:Staking is only compatible with tokens of decimal 1e18

Category	Severity	Client Response	Contributor
Logical	**Medium**	**Fixed**	**Hacker007, 0xffchain**

Code Reference

- code/contracts/staking/SingleStaking.sol#L90-L91
- code/contracts/staking/SingleStaking.sol#L103-L108
- code/contracts/staking/DoubleStaking.sol#L133-L137
- code/contracts/staking/DoubleStaking.sol#L140-L178
- code/contracts/staking/DoubleStaking.sol#L220-L225
- code/contracts/staking/DoubleStaking.sol#L228-L267

```
90:accRewardPerShare += (emitionRate * timePassed);

103:uint256 pending = ((user.amount + user.loyalAmount) * accRewardPerShare) / 1e18 - user.rewardDebt;
104:
105:            if (pending > 0) {
106:                IMintableToken(esTokaToken).mint(msgSender, pending);
107:                emit Harvest(msgSender, pending);
108:            }

133:uint256 timePassed = block.timestamp - pool.lastRewardTime;
134:
135:        pool.accRewardPerShare += (pool.emitionRate * timePassed);
136:
137:        pool.lastRewardTime = block.timestamp;

140:function stake(uint256 _poolId, uint256 _amountA) external {
141:        address msgSender = msg.sender;
142:        PoolInfo storage pool = poolInfo[_poolId];
143:        UserInfo storage user = userInfo[_poolId][msgSender];
144:
145:        updateReward(_poolId);
146:
147:        // If the user had staked before, harvest the reward first
148:        if ((user.amountA + user.loyalAmountA) > 0) {
149:            uint256 pending = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18 - user.rewardDebt;
150:
151:            if (pending > 0) {
152:                IMintableToken(esTokaToken).mint(msgSender, pending);
153:                emit Harvest(msgSender, pending);
154:            }
155:        }
156:
157:        uint256 amountB;
158:        if (_amountA > 0) {
159:            // Fixed ratio of tokenA to tokenB
160:            amountB = (_amountA * pool.stakeRatio) / 1000;
161:
162:            // Transfer tokens to this contract
163:            // The amount of tokens transferred is not cutting the fees
164:            IERC20(pool.tokenA).safeTransferFrom(msgSender, address(this), _amountA);
```

```
165:            IERC20(pool.tokenB).safeTransferFrom(msgSender, address(this), amountB);
166:
167:            user.amountA += _amountA;
168:            user.amountB += amountB;
169:
170:            pool.supplyA += _amountA;
171:            pool.supplyB += amountB;
172:        }
173:
174:        user.lastStakeTime = block.timestamp;
175:        user.rewardDebt = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18;
176:
177:        emit Stake(msgSender, _amountA, amountB);
178:    }

220:uint256 pending = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18 - user.rewardDebt;
221:        if (pending > 0) {
222:            IMintableToken(esTokaToken).mint(msgSender, pending);
223:            emit Harvest(msgSender, pending);
224:        }
225:        user.rewardDebt = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18;

228:function loyalStake(uint256 _poolId, uint256 _amountA) external {
229:        address msgSender = msg.sender;
230:        PoolInfo storage pool = poolInfo[_poolId];
231:        UserInfo storage user = userInfo[_poolId][msgSender];
232:
233:        updateReward(_poolId);
234:        if ((user.amountA + user.loyalAmountA) > 0) {
235:            uint256 pending = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18 - user.rewardDebt;
236:
237:            if (pending > 0) {
238:                IMintableToken(esTokaToken).mint(msgSender, pending);
239:                emit Harvest(msgSender, pending);
240:            }
241:        }
242:
243:        uint256 amountB;
244:        if (_amountA > 0) {
```

```
            amountB = (_amountA * pool.stakeRatio) / 1000;

            IERC20(pool.tokenA).safeTransferFrom(msgSender, address(this), _amountA);
            IERC20(pool.tokenB).safeTransferFrom(msgSender, address(this), amountB);

            user.loyalAmountA += _amountA;
            user.loyalAmountB += amountB;

            pool.loyalSupplyA += _amountA;
            pool.loyalSupplyB += amountB;

            uint256 pending = (_amountA * pool.instantRewardRate) / 1000;

            IMintableToken(esTokaToken).mint(msgSender, pending);
            emit LoyalHarvest(msgSender, pending);
        }

        user.lastLoyalStakeTime = block.timestamp;
        user.rewardDebt = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18;

        emit LoyalStake(msgSender, _amountA, amountB);
    }
```

Description

Hacker007 : The contract `DoubleStaking` uses the following formal to calculate the reward.

```
        uint256 pending = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18 - use
r.rewardDebt;
        if (pending > 0) {
            IMintableToken(esTokaToken).mint(msgSender, pending);
            emit Harvest(msgSender, pending);
        }
        user.rewardDebt = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18;
```

And `pool.accRewardPerShare` is calculated as below:

```
        uint256 timePassed = block.timestamp - pool.lastRewardTime;

        pool.accRewardPerShare += (pool.emitionRate * timePassed);

        pool.lastRewardTime = block.timestamp;
```

Per the contract `EsTokaToken`, esTokaToken is a token with decimal 18. Consider this case, tokenA is a token with decimal 18 and the emitionRate is 1000(according to test file `DoubleStaking.ts`), and timePassed is 30 days. Alice stakes 1000 tokenA. After 30 days later, Alice starts to harvest token. Following the aforementioned formal, the pending reward is:

```
user.rewardDebt = 1000*1e18*pool.accRewardPerShare1/1e18;
pool.accRewardPerShare2 = pool.accRewardPerShare1 + 1000*30 days =  pool.accRewardPerShare1 + 2.592*
1e9
uint256 pending = ((1000*1e18 + 0) * pool.accRewardPerShare) / 1e18 - user.rewardDebt
                              = 1000*1e18*2.592*1e9/1e18
                              = 2.592*1e12
```

The reward is far less than 1 EsTokaToken. The thing may get even worse(rounding to zero) if `tokenA`'s decimal is small, (e.g. 6 USDT)

The same issue happens in SingleStaking.

0xffchain : The contract `DoubleStaking.sol` is only compactable with tokens of 1e18 decimals, This is problematic as it excludes very liquid markets/tokens from being staked on the contract. It is also worth noting that in the docs and last Twitter space on Tonka.finance is stated that it would expose the protocol to as many liquid alt tokens as possible. An example of a market Tonka would be missing from is the bitcoin market on the EVM, WBTC has a TVL of over $7B and it has a decimal of 1e8, USDC has a TVL of $25B and 1e6 decimal, all on ethereum. BTC is also listed as collataral options on Tonka website, meaning its users are exposed to the bitcoin ecosystem, and WBTC has the highest TVL for a bitcoin wrapper on ethereum.

`user.rewardDebt = ((user.amountA + user.loyalAmountA) * pool.accRewardPerShare) / 1e18;`

It will be advisable to change this hard requirement to allow any liquid token on Ethereum or any EVM compactable chain to be staked on Tonka.

Recommendation

Hacker007 : Some measures can mitigate the issue:

1. Add a multiplier(1e18) when calculating `pool.accRewardPerShare`.
2. Add a decimal conversion between tokenA and `esTokaToken`.
3. Set a proper emitionRate.

   0xffchain : Remove the hard requirement of only staking 1e18 tokens, rather call the tokens decimals like so

`IERC20(token).decimals()` to find out its decimals for each calculation it is required.

Client Response

Fixed.We will set a proper emitionRate to provide users with an APY of about 150%.

TFS-4:Renounce ownership should be disabled

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	crjr0629

Code Reference

- code/contracts/staking/SingleStaking.sol#L227-L235
- code/contracts/staking/DoubleStaking.sol#L300-L307

```
227:function collect(address _token, uint256 _amount, bool _isETH) external onlyOwner {
228:        address msgSender = msg.sender;
229:
230:        if (_isETH) {
231:            payable(msgSender).transfer(_amount);
232:        } else {
233:            IERC20(_token).safeTransfer(msgSender, _amount);
234:        }
235:    }

300:function collect(address _token, uint256 _amount, bool _isETH) external onlyOwner {
301:        address msgSender = msg.sender;
302:
303:        if (_isETH) {
304:            payable(msgSender).transfer(_amount);
305:        } else {
306:            IERC20(_token).safeTransfer(msgSender, _amount);
307:        }
```

Description

crjr0629 : By design all contracts have some access control that rely on the owner of the contract to be able to perform some actions. However, the owner can renounce ownership of the contract, this will make the contract ownerless and the access control will be useless.

Recommendation

crjr0629 : override the function `renounceOwnership()` from the `Ownable` contract to prevent the owner from renouncing ownership.

Client Response

Fixed.fix: disable renounce ownership commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/b8629fcccf56dd7775a4d08bd97e45fecd43f0ea

TFS-5: `call()` should be used instead of `transfer()` on an address payable

Category	Severity	Client Response	Contributor
Language Specific	Low	Fixed	8olidity, 0xffchain, Hacker007

Code Reference

- code/contracts/token/TokaVesting.sol#L112
- code/contracts/staking/SingleStaking.sol#L231
- code/contracts/staking/DoubleStaking.sol#L300-L308
- code/contracts/staking/DoubleStaking.sol#L304

```
112:payable(msgSender).transfer(_amount);

231:payable(msgSender).transfer(_amount);

231:payable(msgSender).transfer(_amount);

300:function collect(address _token, uint256 _amount, bool _isETH) external onlyOwner {
301:        address msgSender = msg.sender;
302:
303:        if (_isETH) {
304:            payable(msgSender).transfer(_amount);
305:        } else {
306:            IERC20(_token).safeTransfer(msgSender, _amount);
307:        }
308:    }

304:payable(msgSender).transfer(_amount);

304:payable(msgSender).transfer(_amount);
```

Description

8olidity : In both of the withdraw functions, transfer() is used for native ETH withdrawal. The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.

0xffchain : When sending ETH, use call() instead of transfer(). The transfer() function only allows the recipient to use 2300 gas and sload opcode already cost 800 gas. If the recipient needs more than that, transfers will fail. In the future gas costs might change increasing the likelihood of that happening. If this happens it means the user can not withdraw its claim causing a possible DOS for the user for that day and thus loosing out on its claim. And if the receiving account is a proxy contract, it might not recieve it correctly.
Hacker007 : The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, using these functions for value transfers has often been recommended to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks, breaking already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase in the SLOAD instruction.

The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract implements a payable fallback that uses more than 2300 gas units.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through a proxy, raising the call's gas usage above 2300.
- Additionally, using more than 2300 gas might be mandatory for some multisig wallets.

Recommendation

8olidity : Use call() instead of transfer().
0xffchain : Use call and not transfer. This is a guarded function restricted to an admin, and the admin is allowed to supply any amount as input, so there is nothing the transfer function protects from that the admin does not have full access to.
Hacker007 : Use `call()` instead of `transfer()` to transfer native tokens.

Client Response

Fixed,using call to replace transfer commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/89f9603cb99097e9b64f2be660d2e11ad6341d78

TFS-6:Missing calls to `__Pausable_init()` function in `TokaVesting`

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	Hacker007

Code Reference

- code/contracts/token/TokaVesting.sol#L38-L43

```
38:function initialize(IEsTokaToken _esTokaToken, address _tokaToken) public initializer {
39:        __Ownable_init(msg.sender);
40:        esTokaToken = _esTokaToken;
41:        tokaToken = _tokaToken;
42:        duration = 90 days; // 90 days
43:    }
```

Description

Hacker007 : The contract `TokaVesting` is an upgradeable contract that inherits from `PausableUpgradeable`, however the function `__Pausable_init()` is missing in the function `initialize()`, which may bring unexpected results to the contract.

Recommendation

Hacker007 : recommend calling `__Pausable_init()` in the function `initialize()`.

```
    function initialize(IEsTokaToken _esTokaToken, address _tokaToken) public initializer {
        __Pausable_init();
        __Ownable_init(msg.sender);
        esTokaToken = _esTokaToken;
        tokaToken = _tokaToken;
        duration = 90 days; // 90 days
    }
```

Client Response

Fixed. fix: add __Pausable_init() function call in TokaVesting commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/8128882bf3869e99af70ce751a1c8825fff87173

TFS-7:Unprotected initializer

Category	Severity	Client Response	Contributor
Logical	**Low**	**Fixed**	**Hacker007**

Code Reference

- code/contracts/token/EsTokaToken.sol#L8
- code/contracts/staking/DoubleStaking.sol#L11
- code/contracts/staking/SingleStaking.sol#L13
- code/contracts/token/TokaVesting.sol#L15

```
8:contract EsTokaToken is ERC20Upgradeable, OwnableUpgradeable {

11:contract DoubleStaking is OwnableUpgradeable {

13:contract SingleStaking is OwnableUpgradeable {

15:contract TokaVesting is OwnableUpgradeable, PausableUpgradeable {
```

Description

Hacker007 : One or more logic contracts do not protect their initializers. An attacker can call the initializer and assume ownership of the logic contract, whereby she can perform privileged operations that trick unsuspecting users into believing that she is the owner of the upgradeable contract.

Recommendation

Hacker007 : We advise calling `_disableInitialize` in the constructor to prevent the function `initialize()` from being called on the logic contract.

Reference: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#initializing_the_implementation_contract

Client Response

Fixed.fix: add _disableInitialize in the construtor commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/b4faf43aa01a8f8341ce5f0a190ccc351c5af67a

TFS-8:Wrong lastStakeTime update logic in `SingleStaking::stake()` and `DoubleStaking ::stake()` function

Category	Severity	Client Response	Contributor
Logical	**Low**	**Fixed**	**Hupixiong3**

Code Reference

- code/contracts/staking/SingleStaking.sol#L111-L122
- code/contracts/staking/DoubleStaking.sol#L158-L174
- code/contracts/staking/SingleStaking.sol#L184-L197
- code/contracts/staking/DoubleStaking.sol#L244-L262

```
111:if (_amount > 0) {
112:            // Fixed ratio of tokenA to tokenB
113:
114:            // Transfer tokens to this contract
115:            // The amount of tokens transferred is not cutting the fees
116:            IERC20(tokaToken).safeTransferFrom(msgSender, address(this), _amount);
117:
118:            user.amount += _amount;
119:            supply += _amount;
120:        }
121:
122:        user.lastStakeTime = block.timestamp;

158:if (_amountA > 0) {
159:            // Fixed ratio of tokenA to tokenB
160:            amountB = (_amountA * pool.stakeRatio) / 1000;
161:
162:            // Transfer tokens to this contract
163:            // The amount of tokens transferred is not cutting the fees
164:            IERC20(pool.tokenA).safeTransferFrom(msgSender, address(this), _amountA);
165:            IERC20(pool.tokenB).safeTransferFrom(msgSender, address(this), amountB);
166:
167:            user.amountA += _amountA;
168:            user.amountB += amountB;
169:
170:            pool.supplyA += _amountA;
171:            pool.supplyB += amountB;
172:        }
173:
174:        user.lastStakeTime = block.timestamp;

184:if (_amount > 0) {
185:            IERC20(tokaToken).safeTransferFrom(msgSender, address(this), _amount);
186:
187:            user.loyalAmount += _amount;
188:
189:            loyalSupply += _amount;
190:
191:            uint256 pending = (_amount * instantRewardRate) / 1000;
192:
193:            IMintableToken(esTokaToken).mint(msgSender, pending);
194:            emit LoyalHarvest(msgSender, pending);
```

```
        }

        user.lastLoyalStakeTime = block.timestamp;

if (_amountA > 0) {
            amountB = (_amountA * pool.stakeRatio) / 1000;

            IERC20(pool.tokenA).safeTransferFrom(msgSender, address(this), _amountA);
            IERC20(pool.tokenB).safeTransferFrom(msgSender, address(this), amountB);

            user.loyalAmountA += _amountA;
            user.loyalAmountB += amountB;

            pool.loyalSupplyA += _amountA;
            pool.loyalSupplyB += amountB;

            uint256 pending = (_amountA * pool.instantRewardRate) / 1000;

            IMintableToken(esTokaToken).mint(msgSender, pending);
            emit LoyalHarvest(msgSender, pending);
        }

        user.lastLoyalStakeTime = block.timestamp;
```

Description

Hupixiong3 : When a value of 0 is passed to stake(), this is invalid stake but updates lastStakeTime, which unreasonably extends the user's withdrawal time.

Recommendation

Hupixiong3 : Optimize stake() logic to prevent errors that prolong user withdrawal time.

Client Response

Fixed. fix: add zero staking amount check commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/bf7c3689abfa06fa884dbf8ed9bc338ff5a6f4b0

TFS-9:Event Vesting should be emitted with vesting amount instead of claimed amount

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	minhquanym

Code Reference

- code/contracts/token/TokaVesting.sol#L87

```
87:emit Vesting(msgSender, amount);
```

Description

minhquanym : In the function `vesting()`, the `Vesting` event is currently emitted using the vested/released amount rather than the vesting amount. This discrepancy might lead to confusion and should be addressed.

```
function vesting(uint256 _amount) external whenNotPaused {
    ...

    uint256 amount = (user.amountReleasePerSec * passedTime) / 1e12 - user.released;
    if (amount > 0) {
        user.released += amount;
        IMintableToken(tokaToken).mint(msgSender, amount);
    }

    esTokaToken.burn(msgSender, _amount);

    user.start = block.timestamp;
    user.amount = user.amount - user.released + _amount;
    user.amountReleasePerSec = (user.amount * 1e12) / duration;
    user.released = 0;

    // @audit Should emit with `_amount` instead of `amount`
    emit Vesting(msgSender, amount);
}
```

Recommendation

minhquanym : Consider emitting `_amount` instead

```
- emit Vesting(msgSender, amount);
+ emit Vesting(msgSender, _amount);
```

Client Response

Fixed, emit _amount instead of amount. commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/c626f3a0e2c4d9c6afe0a527853ea3122e66ba5d

TFS-10:Claim event should be emitted in function `vesting()`

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	minhquanym

Code Reference

- code/contracts/token/TokaVesting.sol#L75-L78

```
75:if (amount > 0) {
76:            user.released += amount;
77:            IMintableToken(tokaToken).mint(msgSender, amount);
78:        }
```

Description

minhquanym : In the function `TokaVesting.vesting()`, it firstly tries to claim any vested token for user before actually updating the new vesting info of user. So in this case, a `Claim` event should also be emitted similarly to how it is emitted in the function `claim()`.

```
function vesting(uint256 _amount) external whenNotPaused {
    ...

    uint256 amount = (user.amountReleasePerSec * passedTime) / 1e12 - user.released;
    if (amount > 0) {
        // @audit should emit Claim event
        user.released += amount;
        IMintableToken(tokaToken).mint(msgSender, amount);
    }

    esTokaToken.burn(msgSender, _amount);

    user.start = block.timestamp;
    user.amount = user.amount - user.released + _amount;
    user.amountReleasePerSec = (user.amount * 1e12) / duration;
    user.released = 0;

    emit Vesting(msgSender, amount);
}
```

Recommendation

minhquanym : Consider emit Claim event after releasing `tokaToken` in the function `vesting()`.

Client Response

Fixed,add emitting Claim event in vesting func commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/95d647bdab373c50e45607d712db6ee529486349

TFS-11:Supply cap is not enforced in `TokaToken`

Category	Severity	Client Response	Contributor
Logical	**Informational**	**Fixed**	**minhquanym**

Code Reference

- code/contracts/token/TokaToken.sol#L12

```
12:uint256 public cap = 500_000_000;
```

Description

minhquanym : In the `TokaToken` contract, there exists a supply cap of 500 million token wei. Unfortunately, this cap is not being verified anywhere within the contract.

```
uint256 public cap = 500_000_000; // @audit cap is not enforced in contract
...

function mint(address _to, uint256 _amount) external {
    require(isMinter[msg.sender], "Only minter can mint");
    _mint(_to, _amount);
}
```

Recommendation

minhquanym : Consider adding a supply cap check in the function `mint()`.

Client Response

Fixed,fix in the commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/7673949e72d3139ffd63310ab288a3ffa04d5463

TFS-12:Missing Zero Address Check

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	8olidity, helookslikeme, LiRiu

Code Reference

- code/contracts/token/TokaToken.sol#L16-L40
- code/contracts/token/EsTokaToken.sol#L57-L89
- code/contracts/token/TokaVesting.sol#L63
- code/contracts/token/TokaVesting.sol#L108

```
function addMinter(address _minter) external onlyOwner {
        isMinter[_minter] = true;
    }

    function removeMinter(address _minter) external onlyOwner {
        isMinter[_minter] = false;
    }

    function addBurner(address _burner) external onlyOwner {
        isBurner[_burner] = true;
    }

    function removeBurner(address _burner) external onlyOwner {
        isBurner[_burner] = false;
    }

    function mint(address _to, uint256 _amount) external {
        require(isMinter[msg.sender], "Only minter can mint");
        _mint(_to, _amount);
    }

    function burn(address _from, uint256 _amount) external {
        require(isBurner[msg.sender], "Only burner can burn");
        _burn(_from, _amount);
    }

function addMinter(address _minter) external onlyOwner {
        isMinter[_minter] = true;
    }

    function removeMinter(address _minter) external onlyOwner {
        isMinter[_minter] = false;
    }

    function addBurner(address _burner) external onlyOwner {
        isBurner[_burner] = true;
    }

    function removeBurner(address _burner) external onlyOwner {
        isBurner[_burner] = false;
    }

    function mint(address _to, uint256 _amount) external {
        require(isMinter[msg.sender], "Only minter can mint");
        _mint(_to, _amount);
    }

    function burn(address _from, uint256 _amount) external {
        require(isBurner[msg.sender], "Only burner can burn");
        _burn(_from, _amount);
    }

    function addWhitelist(address _account) external onlyOwner {
        whitelist[_account] = true;
    }

    function removeWhitelist(address _account) external onlyOwner {
        whitelist[_account] = false;
    }

function vesting(uint256 _amount) external whenNotPaused {

function collect(address _token, uint256 _amount, bool _isETH) external onlyOwner {
```

Description

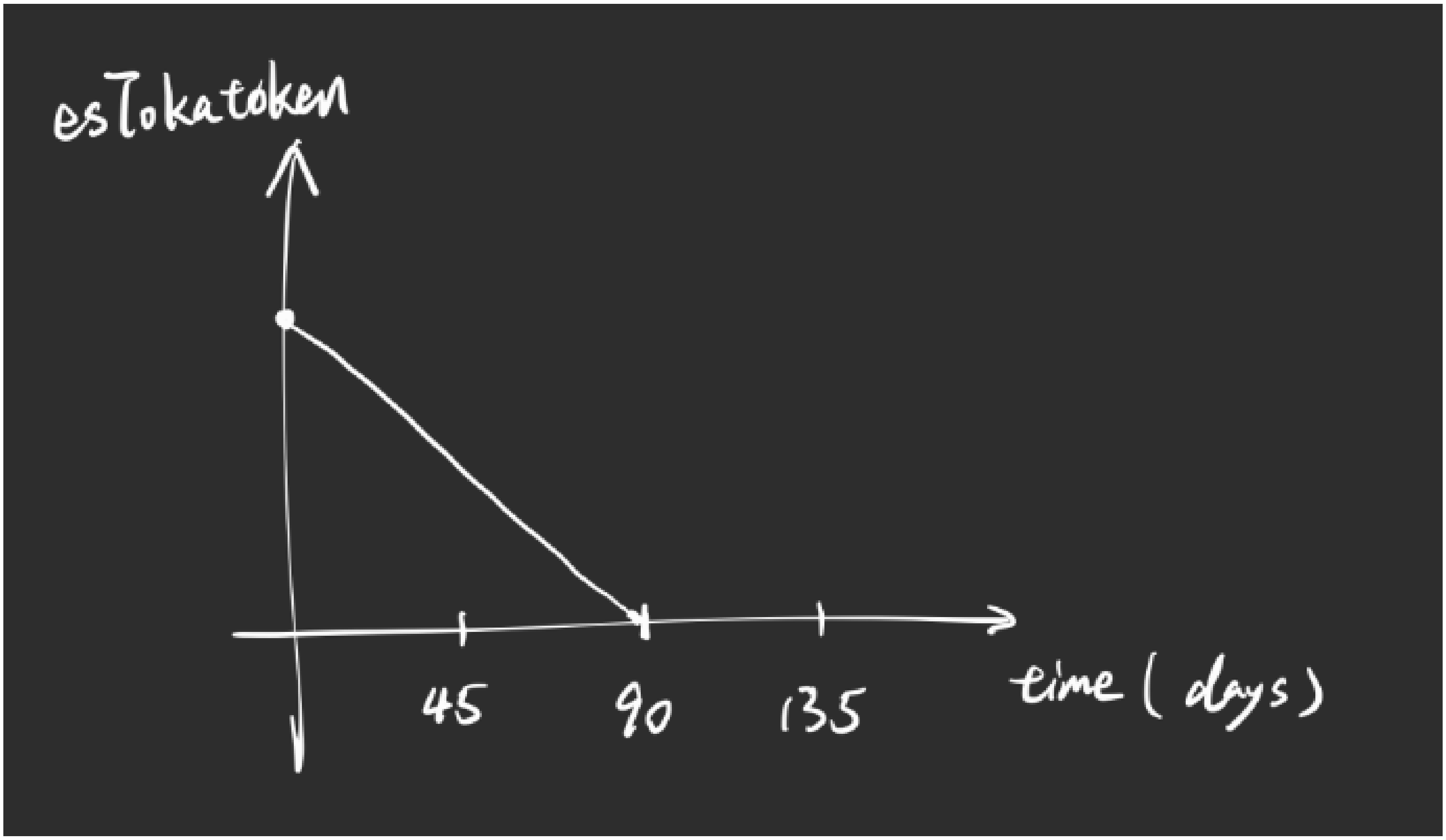
8olidity :

```
// code/contracts/token/EsTokaToken.sol
    function addMinter(address _minter) external onlyOwner {
        isMinter[_minter] = true;
    }
    function removeMinter(address _minter) external onlyOwner {
        isMinter[_minter] = false;
    }
    function addBurner(address _burner) external onlyOwner {
        isBurner[_burner] = true;
    }
    function removeBurner(address _burner) external onlyOwner {
        isBurner[_burner] = false;
    }
    function addWhitelist(address _account) external onlyOwner {
        whitelist[_account] = true;
    }
    function removeWhitelist(address _account) external onlyOwner {
        whitelist[_account] = false;
    }




// code/contracts/staking/SingleStaking.sol

    function initialize(address _esTokaToken, address _tokaToken) public initializer {
        __Ownable_init(msg.sender);
        esTokaToken = _esTokaToken;
        withdrawalCooldown = 1_209_600; // 14 days
        loyalCooldown = 126_144_000; // 4 years
    }
// code/contracts/token/TokaToken.sol
    function addMinter(address _minter) external onlyOwner {
        isMinter[_minter] = true;
    }
    function removeMinter(address _minter) external onlyOwner {
        isMinter[_minter] = false;
    }
    function addBurner(address _burner) external onlyOwner {
        isBurner[_burner] = true;
    }
    function removeBurner(address _burner) external onlyOwner {
        isBurner[_burner] = false;
```

```
  }
```

helookslikeme : _token does not verify whether it is a blacklist or 0 address

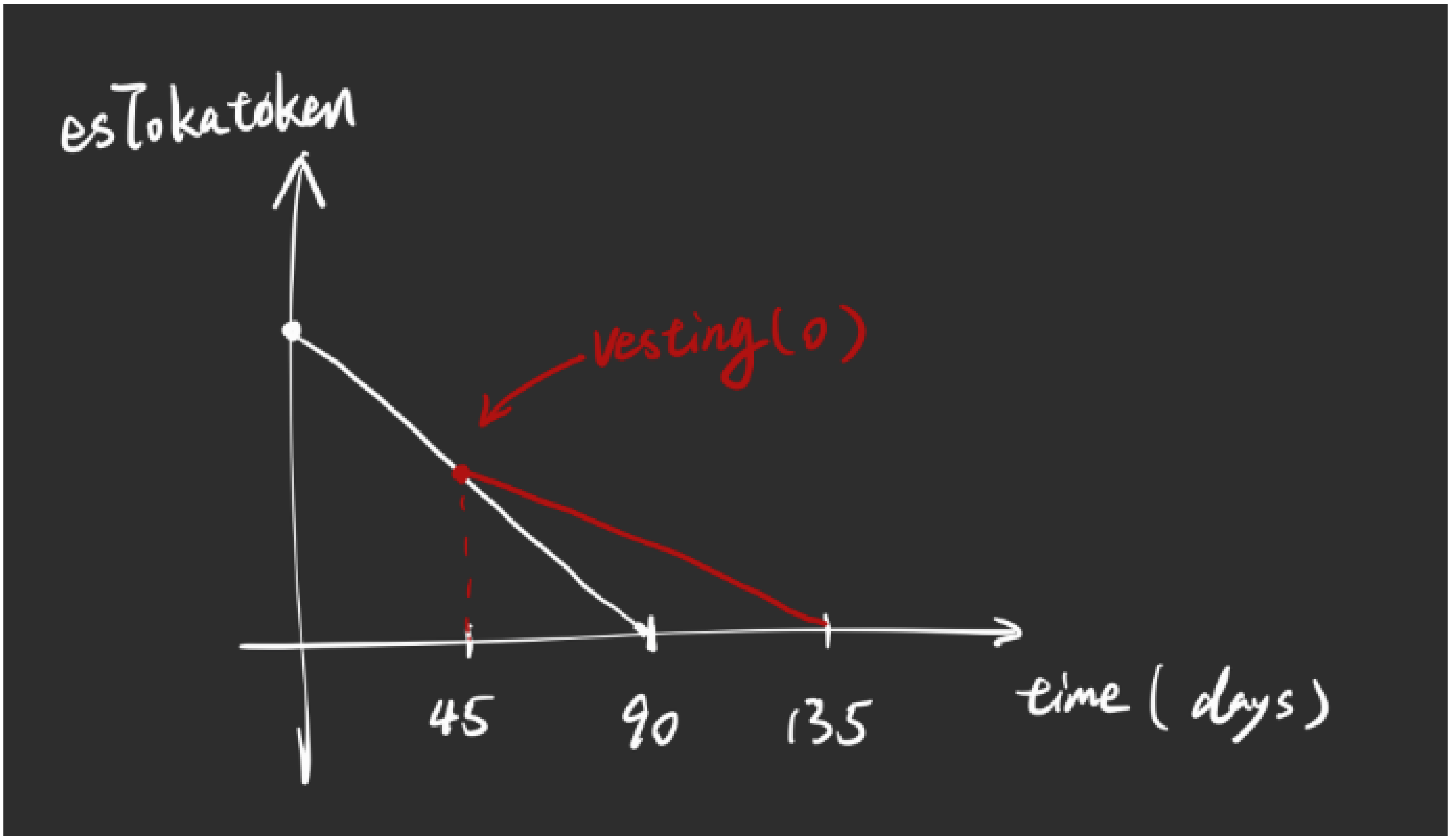
LiRiu : The vesting function allows users to burn esTokaToken and receive an equivalent amount of TokaToken within three months.



The variable that records the user's release rate in the function is `user.amountReleasePerSec`. This variable is updated every time the vesting function is called.

```
        user.amount = user.amount - user.released + _amount;
        user.amountReleasePerSec = (user.amount * 1e12) / duration;
        user.released = 0;
```

Due to the function not checking that _amount is not zero, it may cause users to be unable to receive profits in a timely manner.



Recommendation

8olidity : Add check of zero address in important operation.
helookslikeme : _token ！ = address （0）
LiRiu : I understand that if _amount == 0, the vesting will degrade into the claim function, which would be more in line with the design expectations.

But this contract would be too complicated.

I think a more concise solution is to add `require( _amount != 0, "_amount should not be 0.")` in the vesting function.

Client Response

Fixed,add zero address check commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/b9d0c47bc03bfd67c9630e417d3b222c34b08319

TFS-13:redundant variables `cap`

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	8olidity, Hupixiong3, Hacker007, crjr0629

Code Reference

- code/contracts/token/TokaToken.sol#L12
- code/contracts/token/TokaToken.sol#L12-L13

```
12:uint256 public cap = 500_000_000;

12:uint256 public cap = 500_000_000;

12:uint256 public cap = 500_000_000;
```

Description

8olidity : `cap` is a redundant variable, it is defined but not used.

```
contract TokaToken is ERC20, Ownable {
    mapping(address => bool) public isMinter;
    mapping(address => bool) public isBurner;

    uint256 public cap = 500_000_000;
```

Hupixiong3 : If unused code snippets are useful, they need to be completed logically. If they are not useful, they need to be deleted to prevent the overall code from being affected.
Hacker007 : The state variable `cap` is defined but not used, which seems to be redundant.
crjr0629 : Contract `TokaToken.sol` has an unused variable cap, it is not used anywhere else on the code.

Recommendation

8olidity : Delete this variable
Hupixiong3 : Complete logic or remove redundant code.
Hacker007 : Remove the unused state variable `cap`.
crjr0629 : consider removing the variable cap.

Client Response

Fixed,fix in the commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/7673949e72d3139ffd63310ab288a3ffa04d5463

TFS-14:Unified optimization of the coinage process in `TokaVesting::claim()`

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	Hupixiong3

Code Reference

- code/contracts/token/TokaVesting.sol#L99-L103

```
99:uint256 amount = (user.amountReleasePerSec * passedTime) / 1e12 - user.released;
100:
101:        user.released += amount;
102:
103:        IMintableToken(tokaToken).mint(msgSender, amount);
```

Description

Hupixiong3 : The claim() function mint token logic should be consistent with the vesting() function, needs to determine the number of amount to prevent invalid operations.

Recommendation

Hupixiong3 : Optimizes mint token logic for claim() function.

```
        uint256 amount = (user.amountReleasePerSec * passedTime) / 1e12 - user.released;
        if (amount > 0) {
            user.released += amount;
            IMintableToken(tokaToken).mint(msgSender, amount);
        }
```

Client Response

Fixed.fix: optimise transfer token logic for claim func commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/8a343c4b25dfbc17161f3248ef06a2fad3dbf220

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.