wisdomise

SMART CONTRACTS REVIEW

zokyo

February 27th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING WISDOMISE

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points=
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 5 Medium issues: 4 resolved and 1 invalid issues = 0 points deducted
- 7 Low issues: 7 resolved issues = 0 points deducted
- 6 Informational issues: 4 resolved and 2 acknowledged issues = - 1 points deducted

Thus, 100 - 1 = 99

TECHNICAL SUMMARY

This document outlines the overall security of the Wisdomise smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Wisdomise smart contracts codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Wisdomise team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Wisdomise repository:
Repo: https://github.com/Wisdomise/wsdm-token

Last commit - 2e807b0ab18b0ff61697b0d9bf7506f20364c9ff

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./locking.sol
- ./TokenMigration.sol
- ./TokenDistributor.sol
- ./wsdm.sol
- ./vesting.sol
- ./interfaces/ITokenDistributor.sol
- ./MerkleDistributorWithDeadline.sol
- ./MerkleDistributor.sol
- ./interfaces/IMerkleDistributor.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Wisdomise smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contracts by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Wisdomise team and the Wisdomise team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Strict equality in migrate() method	Medium	Resolved
2	Investor can be front running the procedure in _addInvestor	Medium	Invalid
3	Emergency Exit Logic Undermines Pausable Mechanism	Medium	Resolved
4	Use of SafePermit is removed Based on recommendations from @trust1995	Medium	Resolved
5	Use Ownable instead of an immutable beneficiary in vesting.sol	Medium	Resolved
6	Precision Loss due to truncated decimals	Low	Resolved
7	Penalty fee can be higher than the user balance	Low	Resolved
8	No address(0) validation for constructor parameters	Low	Resolved
9	Records can be overridden without checking	Low	Resolved
10	hasUnlockedBefore is not reset back to false on cancelUnlock	Low	Resolved
11	Centralization risk	Low	Resolved
12	Lack of Event Emission in Privileged Functions	Low	Resolved
13	Short-duration logic can be manipulated by miners.	Informational	Resolved
14	Recommendation for Upgradeable ERC20 Token Implementation	Informational	Acknowledged
15	Gas Consumption Concerns Due to Extended Error Messages	Informational	Acknowledged
16	Unnecessary Safe Math is being utilized	Informational	Resolved
17	Floating Pragma Version in Solidity Files	Informational	Resolved
18	Rename mapping	Informational	Resolved

Strict equality in migrate() method

Description :

In Contract TokenMigration.sol, the method migrate() allows users to migrate their TWSDM tokens to WSDM tokens.

This method contains the following logic:

```
        uint256 angelInvestorShares = angelInvestors[msg.sender];
        uint256 strategicInvestorShares =
strategicInvestors[msg.sender];
        require(
            balance == angelInvestorShares + strategicInvestorShares,
"TokenMigration: twsdm balance does not match with investor shares"
        );
```

If the balance is not equal to the investor shares, it will fail to migrate the tokens. Anyone can front-run a user's migrate() txn to transfer 1 wei of TWSDM to the user account. This will make the txn revert because of the above `require` statement.

This will lead to DoS attacks for users especially users with huge amounts of TWSDM tokens and deny them migrating their tokens. Given that these contracts are deployed on Polygon, the cost of the attack will be less as well.

Recommendation:
Update the migrate() logic to handle the case when the balance is more than the investor shares as well. Also, revert in case investor shares are `0` for msg.sender.

Investor can be front running the procedure in _addInvestor

Description :

This can take place in case owner intentionally attempts to overwrite a previous assignment for an investor. Investor can capitalize on that and seize both amounts.
That is described in the following scenario:
Scenario:
txn.a1: _addInvestor(0xB0B, 100, ANGEL)
txn.a2: _addInvestor(0xB0B, 50, ANGEL) // Admin realized a mistake and intended to overwrite the quantity
txn.b1: migrate() // called by 0xB0B
txn.b2: migrate() // called by 0xB0B

- 0xB0B tunes the gas payment in order to squeeze txn.b1 in between txn.a1 and txn.a2. Therefore 0xB0B receives 100 then from txn.b2 he receives 50 which is the fair amount intended to be assigned by admin but 0xB0B ends up capitalizing on that scenario to take both amounts.

This issue should be addressed along side: Records can be overridden without checking & Strict equality in migrate() method. Refactoring of the migration logic is recommended.

Recommendation:
This issue is similar to the ERC20 approval frontrunning which has been solved by introducing increaseAllowance and decreaseAllowance in later versions of ERC20, therefore a similar mechanism helps avoid this. It is recommended though to address that issue alongside the 2 other issues and refactor the logic accordingly.

Fix - The manual setting of shares by the admin is only meant to allocate strategic and angel shares. The balance of twsdm remains the single source of truth. Function migrate() burns all the balance of the caller and it can only be called once.

Emergency Exit Logic Undermines Pausable Mechanism

Description :

The contract Locking implements an emergency exit mechanism alongside the Pausable functionality inherited from OpenZeppelin's Pausable contract. However, the logic used to determine whether actions can proceed during an emergency or paused state could potentially compromise the security and intended functionality of the contract.

```
modifier whenNotPausedOrEmergencyExitActive() {
    require(!paused() || emergencyExit, "Locking: Paused and Emergency Exit is deactivated");
    _;
}
```

This line is part of the whenNotPausedOrEmergencyExitActive modifier, which is intended to restrict certain contract functionalities when the contract is paused, except in cases of an emergency exit. The issue arises from the logical OR (||) operator, which allows actions to proceed if either the contract is not paused (!paused()) or if the emergency exit is active (emergencyExit). This means that during an emergency (when emergencyExit is set to true), operations can still be executed even if the contract is paused.
This logic undermines the purpose of the emergency stop mechanism, which is to halt operations in the face of detected vulnerabilities or ongoing attacks. Allowing operations to proceed during an emergency, even when the contract is paused, could open up avenues for exploitation, especially if the emergency exit mechanism is not tightly controlled or if the pause state was specifically activated to prevent certain actions from being taken.

Recommendation:

- Create separate functions for actions that should only be available during an emergency. This clear separation ensures that only intended actions are executable in an emergency scenario, reducing the risk of exploitation.
- Redefine the whenNotPausedOrEmergencyExitActive modifier to ensure it strictly enforces the intended restrictions. The modifier should be designed to prevent any operations from proceeding when the contract is paused, with the only exceptions being the newly defined emergency functions.
- For the emergency functions, implement strict access controls to ensure that only authorized entities can activate or execute these functions.

```
contract Locking is Pausable {
    bool private emergencyExit = false;

    modifier whenNotPausedOrEmergencyExitActive() {
        require(!paused() || emergencyExit, "Contract is not paused
and not in emergency");
        _;
    }

    function activateEmergencyExit() public onlyOwner {
        emergencyExit = true;
    }
```

```
    function emergencyFunction() public
whenNotPausedOrEmergencyExitActive {
        // Emergency function logic here
    }

    function normalFunction() public whenNotPaused {
        // Normal function logic here
    }
}
```

Use of SafePermit is removed Based on recommendations from @trust1995

Description :

In the latest openzeppelin-contracts safePermit has been removed

https://github.com/OpenZeppelin/openzeppelin-contracts/pull/4582

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/192e873fcb5b1f6b4b9efc177be231926e2280d2/CHANGELOG.md

Recommendation:
Create custom permit operation (considering all the drawbacks of traditional permit operation) like
There are two important considerations concerning the use of `permit`. The first is that a valid permit signature expresses an allowance, and it should not be assumed to convey additional meaning. In particular, it should not be considered as an intention to spend the approval in any specific way. The second is that because permits have built-in replay protection and can be submitted by anyone, they can be frontrun. A protocol that uses permits should take this into consideration and allow a `permit` call to fail. Combining these two aspects, a good pattern that can be generally recommended is:

try token.permit(msg.sender, spender, value, deadline, v, r, s) {} catch {}

Observe that: 1) `msg.sender` is used as the owner, leaving no ambiguity as to the signer intent, and 2) the use of `try/catch` allows the permit to fail and makes the code tolerant to frontrunning. In general, `address(this)` would be used as the `spender` parameter. Additionally, note that smart contract wallets (such as Argent or Safe) are not able to produce permit signatures, so contracts should have entry points that don't rely on permit.

Use Ownable instead of an immutable beneficiary in vesting.sol

Description :

In the current implementation of the VestingWallet, the beneficiary is set at contract deployment and cannot be altered. This design choice limits the ability to transfer unvested tokens, potentially creating issues if the original beneficiary address becomes compromised or if there's a legitimate need to transfer the vesting schedule to a new address.

Use `Ownable` in `VestingWallet` instead of an immutable beneficiary by ernestognw · Pull Request #4508 · Open Zeppelin/open zeppelin-contracts (github.com)

Recommendation:

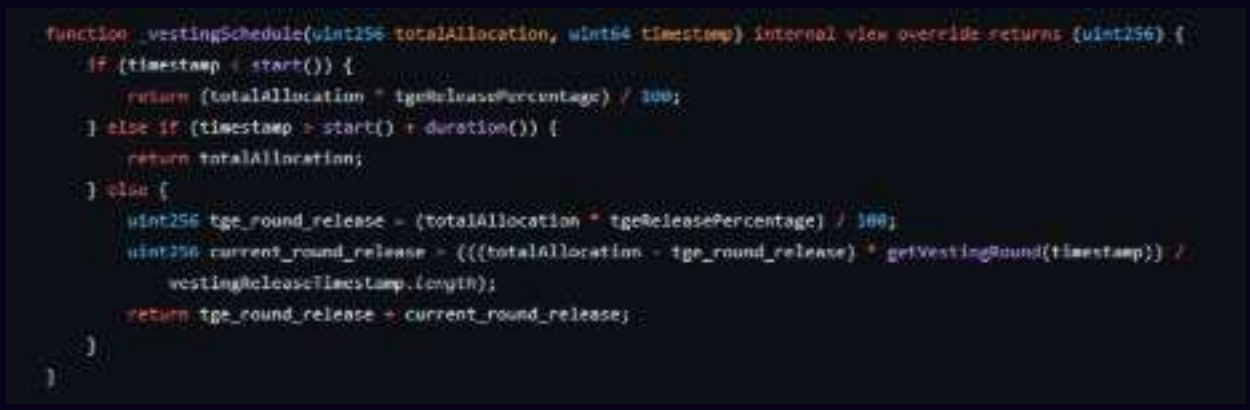
- Inherit from Ownable2Step or Ownable
- Restrict the release() Function

Precision Loss due to truncated decimals

Description :

This line will cause precision loss, however it mainly depends on the constructor arguments. This bug is due to the lack of support for floating-point arithmetic, which means Solidity can only handle integer values.

uint256 tge_round_release = (totalAllocation * tgeReleasePercentage) / 100;



Recommendation:

Consider scaling up the values involved in the calculation before performing arithmetic operations. For example, using a higher base for percentages (e.g., representing 1% as 10000 in calculations instead of 1)

Penalty fee can be higher than the user balance

Description :

In Contract locking.sol, the method unlock allows users to unlock their locked tokens but a penalty fee is incurred on them.

The amount fee is calculated as follows:

```
   function calculatePenalty(address locker, uint256 balance,
uint256 unlockTimestamp) public view returns (uint256) {
        if (emergencyExit) {
            return 0;
        }
        return (balance * calculatePenaltyFee(locker,
unlockTimestamp)) / 10 ** 6;
    }
```

Here if the method calculatePenaltyFee returns a value > 10^6, then the penalty fee will be more than the balance itself. This will revert the user's unlock txn.

It is not validated anywhere that calculatePenaltyFee() will be less than 10^6 always.

Recommendation:
Add a check to ensure that the max penalty fee is equal to the balance of the user locked.

LOW-3 | RESOLVED

No address(0) validation for constructor parameters

Description :

In Contract wsdm.sol, the constructor mints the total supply to the reserve address. It is not validated if that address is address(0) or not. Also no other method exists to set reserve again so if reserve is set address(0) then contract will need to be deployed again.

In Contract TokenDistributor.sol, the constructor does not validate if the wsdmTokenAddress is address(0) or not. Also no other method exists to set wsdmTokenAddress.

In Contract TokenMigration.sol, none of the constructor parameters is validated for address(0). Also no other method exists to set these constructor parameters.

Recommendation:
Add the required validation for the above-mentioned contracts.

Records can be overridden without checking

Description :

In function _addInvestor(), the values inside mappings angelInvestors and strategicInvestors can be overridden. Suppose a scenario in which the owner (i.e. procedure can only be triggered by owner) executes the function on same investor with a different balance value, the values are overridden.

```
 function _addInvestor(address investor, uint256 balance, InvestorType investorType)
private {
        require(investor != address(0), "TokenMigration: investor should not be address zero");

        if (investorType == InvestorType.ANGEL) {
            angelInvestors[investor] = balance;
        } else if (investorType == InvestorType.STRATEGIC) {
            strategicInvestors[investor] = balance;
        }

        emit InvestorInserted(investor, balance, investorType);
    }
```

This could be a feature because of the strict equality check imposed in function migrate:

```
require(
    balance == angelInvestorShares + strategicInvestorShares,
        "TokenMigration: twsdm balance does not match with investor shares"
);
```

It is worth noting that the substance of the issue comes from the choice to make the process manual to a big extent. Attention is needed in this part since it might introduce a non-negligble impact if human error arises.

Overridding those mappings introduce the contracts to an attack vector that is explained in a separate finding.

Recommendation:
Refactor the logic of migration putting in mind less human involvement in determining the values.

Fix - The manual setting of shares by the admin is only meant to allocate strategic and angel shares. The balance of twsdm remains the single source of truth. Client is adopting certain methods in order to allocate the shares minimizing the chance of the need to call the function again and override the allocations.

LOW-5 | RESOLVED

hasUnlockedBefore is not reset back to false on cancelUnlock

Description :

When users invoke unlock() to start a process to unlock their assets. The state of unlockedUsers[msg.sender] and _lockedUsers[msg.sender] is changed accordingly. In case users change their mind by calling cancelUnlock() the changes are reset back to the state before calling unlock(). There is an issue specifically in the following:

_lockedUsers[msg.sender].hasUnlockedBefore = true;

Since on calling cancelUnlock() the state before the unlock() is expected to be retrieved. But _lockedUsers[msg.sender].hasUnlockedBefore is not becoming false as it should. This affects the calculation of calculatePenaltyFee() as in that case the value of hasUnlockedBefore would be true changing the course of how penalty is calculated.

Recommendation:
Assign the attribute back to false.
_lockedUsers[msg.sender].hasUnlockedBefore = false;

Fix: Considered as a required spec by the developer team.

Centralization risk

Location: locking.sol, TokenDistributor.sol, TokenMigration.sol

Description :

The smart contracts employ modifiers (e.g. onlyOwner) in functions responsible for carrying out important actions (e.g. setEmergencyExit, addBulkAngelInvestor, ...etc). This approach centralizes control, allowing a single owner to exclusively perform critical actions, that involves asset transfer, posing a risk to decentralization principles.

Risk: Single Point of Failure: Centralized control introduces a single point of failure, where the compromise of the owner's account can lead to unauthorized access and manipulation of critical functions.

Recommendation:
To mitigate the centralization risk, it is recommended to:

Implement Access Control Lists (ACL): Utilize Access Control Lists to assign different roles with specific permissions, allowing for a more granular control structure.

Multi-Sig or Governance Contracts: Consider implementing multi-signature schemes or governance contracts to distribute decision-making authority, reducing the risk of a single point of failure. Multi-Sig can be utilized in the project after deployment without altering the

LOW-7 | RESOLVED

Lack of Event Emission in Privileged Functions

Location: locking.sol, TokenDistributer.sol

Description :

During the security audit of the smart contract, an issue has been identified in the implementation of privileged functions, one example is function setNumberOfFreeTrialPeriod(). The codebase has not incorporated event emission within several of these privileged functions. The absence of events in these operations limits transparency, hindering the ability to monitor and track critical changes to the contract's state.

Recommendation:
Emit the relevant events on calling privileged functions.

Fix: Addressed and fixed in commit `2e807b0ab18b0ff61697b0d9bf7506f20364c9ff`.

INFORMATIONAL-1 | RESOLVED

Short-duration logic can be manipulated by miners.

Description:
block.timestamp is utilized in several crucial operations, notably in the calculatePenalty and calculatePenaltyFee functions, which determine the penalty amount based on the duration a user's tokens have been locked. Since block.timestamp is used to calculate unlocking periods and penalties, there's a risk that miners could manipulate timestamps to influence these calculations. Although the Ethereum protocol enforces that each block's timestamp must be greater than the previous block's timestamp and allows only a small future time drift (around 15 seconds), there still exists a window for manipulation, especially in operations sensitive to time adjustments.

```
function calculatePenalty(address locker, uint256 balance, uint256 unlockTimestamp) public view returns (uint256) {
    if (emergencyExit) {
        return 0;
    }
    return (balance * calculatePenaltyFee(locker, unlockTimestamp)) / 10 ** 6;
}

function calculatePenaltyFee(address locker, uint256 unlockTimestamp) public view returns (uint256) {
    uint256 lockTimestamp = lockedUsers[locker].startTimestamp;
    uint256 _currentStep = (unlockTimestamp - lockTimestamp) / MONTHLY_PENALTY_INTERVAL;

    uint256 _penaltyFee;

    uint256 _completeYears = _currentStep / 12;
    uint256 _startFreeDays = _completeYears * (360 days) + lockTimestamp;
    uint256 _endFreeDays = _startFreeDays + freeUnlockDuration;

    _currentStep = _currentStep % 12;

    if (_completeYears > 0 && _startFreeDays < unlockTimestamp && unlockTimestamp < _endFreeDays) {
        _penaltyFee = 0;
    } else if (
        _completeYears == 0 &&
        lockedUsers[locker].hasUnlockedBefore == false &&
        currentStep < numberOfFreeTrialPeriod
    ) {
        _penaltyFee = 0;
    } else {
        _penaltyFee = monthlyPenaltyConfigs[lockedUsers[locker].configId][currentStep];
    }

    return _penaltyFee;
}
```

Recommendation:

- Use External Time Oracles
- Introduce Grace Periods or buffers
- Increase Time Intervals (for functions that are not time-sensitive to the second, use hours or days)

Recommendation for Upgradeable ERC20 Token Implementation

Description:
During the security audit, it is noted that client possess a proactive approach to maintaining and evolving their projects which led them to decide migrate ERC20 token. To further enhance the project's flexibility and upgradability, it is recommended to consider the implementation of an upgradeable version of the ERC20 token standard.

While the current ERC20 implementation is sound and secure, adopting an upgradeable model demonstrates a commitment to ongoing development, adaptability to future changes, and an improved user experience by minimizing migration requirements. The proactive consideration of upgradeability aligns with industry best practices and ensures the project remains resilient and responsive to evolving standards and user needs.

Rationale:
Flexibility for Future Enhancements: An upgradeable ERC20 token allows the development team to introduce improvements, bug fixes, or additional features without requiring users to migrate to a new contract. This flexibility is particularly valuable for long-term projects.

Reduced Migration Overheads: Implementing upgradeability reduces the need for token migrations, minimizing the operational and user experience overheads associated with deploying entirely new contracts.

Adaptability to Changing Requirements: The crypto landscape evolves, and new standards or security practices may emerge. An upgradeable ERC20 token positions the project to adapt more seamlessly to any future changes or advancements in the Ethereum ecosystem.

Recommendation:
Consider implementing an upgradeable ERC20 token using established patterns such as the OpenZeppelin Upgrades Plugins or other industry-recognized upgradeable contract frameworks.

INFORMATIONAL-3 | ACKNOWLEDGED

Gas Consumption Concerns Due to Extended Error Messages

Location: locking.sol, TokenDistributor.sol, TokenMigration.sol

Description:
The smart contract employs detailed error messages within require statements, offering developers and users specific insights into the conditions that trigger a requirement failure. However, the extensive length of these error messages result in heightened gas consumption.

Recommendation:

- Employ shorter concise error messages.
- Use Custom Error objects available from solidity 0.8.0.

INFORMATIONAL-4 | RESOLVED

Unnecessary Safe Math is being utilized

Location: TokenDistributor.sol

Description:
The default safe math operation in solidity versions ^0.8.x incurs extra gas overhead due to it requiring more computation. The following operation, that is being carried out on the iterator of the for-loop, can be more gas-efficient by using the unchecked statement.

In function addBulkPayeeByOwner(), we have:

```
    for (uint256 i = 0; i < accounts.length; i++) {
        _addPayee(accounts[i], shares[i]);
    }
```

As well in TokenMigration.sol, in function _addBulkInvestor().

```
  for (uint256 i = 0; i < investors.length; i++) {
        _addInvestor(investors[i], balances[i], investorType);
    }
  }
```

While the code snippet correctly ensures that the addition operation will not result in an overflow, the unnecessary default safe math usage can be optimized for gas efficiency. Wrapping the operation in an unchecked statement is a recommended practice for situations where the developer can guarantee that overflows/underflows will not occur. This enhancement contributes to more efficient gas utilization without compromising safety.

Recommendation:
Wrap Operation in unchecked Statement, given that the condition preceding the operation ensures there is no risk of overflow. It is a common pattern in for-loops since 0.8.x to be as follow:

```
for (uint256 i = 0; i < length;) {
    ...
    unchecked {
       i++;
    }
}
```

Fix: Addressed and fixed in commit 2e807b0ab18b0ff61697b0d9bf7506f20364c9ff.

INFORMATIONAL-5 | RESOLVED

Floating Pragma Version in Solidity Files

Location: All

Description:
The Solidity files in this codebase contain pragma statements specifying the compiler version in a floating manner (e.g., `^0.8.0`). Floating pragmas can introduce unpredictability in contract behavior, as they allow automatic adoption of newer compiler versions that may introduce breaking changes.

Recommendation:
To ensure stability and predictability in contract behavior, it is recommended to:

Specify Fixed Compiler Version: Instead of using a floating pragma, specify a fixed compiler version to ensure consistency across different deployments and prevent the automatic adoption of potentially incompatible compiler versions.

INFORMATIONAL-6 | RESOLVED

Rename mapping

Description:
In Contract TokenMigration.sol, the mapping angelInvestors and strageticInvestors store the user's shares. It is advised to rename these mappings to angelInvestorShares and strageticInvestorShares for better readability.

Recommendation:
Make the suggested changes.

	./locking.sol **./TokenMigration.sol** **./TokenDistributor.sol** **./wsdm.sol** **./vesting.sol** **./interfaces/ITokenDistributor.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Wisdomise team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Wisdomise team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo