ZAP.

SMART CONTRACTS REVIEW

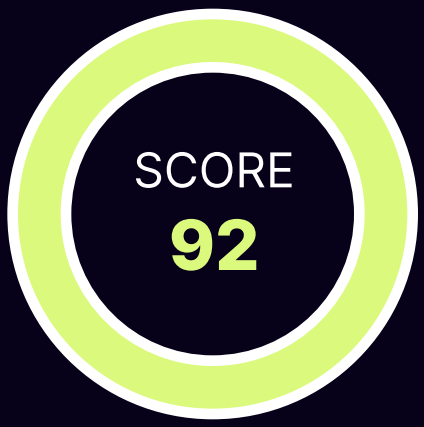
zokyo

April 11th 2024 | v. 1.0

Security Audit Score

PASS

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



ZOKYO AUDIT SCORING ZAP

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:
- 2 Critical issues: 2 resolved = 0 points deducted
- 4 High issues: 4 resolved = 0 points deducted
- 3 Medium issues: 3 resolved = 0 points deducted
- 11 Low issues: 10 resolved and 1 acknowledged = - 3 points deducted
- 9 Informational issues: 8 resolved and 1 partially resolved = 0 points deducted
- lack of test coverage = - 5 points deducted

Thus, 100 - 3 - 5 = 92

TECHNICAL SUMMARY

This document outlines the overall security of the ZAP smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the ZAP smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 2 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 contract/s 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 ZAP team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

Auditing Strategy and Techniques Applied 5

Executive Summary 7

Structure and Organization of the Document 8

Complete Analysis 9

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the ZAP repository:
Repo: https://github.com/Lithium-Ventures/zap-contracts-labs

Last commit: e5cb23a4ab142e25e73840cd12fe7b792ef5df69

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

- Vesting
- VestingFactory
- Admin
- TokenSale
- ERC20 Factory
- SimpleERC20

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 ZAP smart contract/s. 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 contract/s 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 ZAP team and the ZAP 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	Unrestricted call to destroyInstance	Critical	Resolved
2	Reentrance can lead to loss of funds	Critical	Resolved
3	Incorrect Return Of userMaxAllc	High	Resolved
4	Incorrect decimals precision	High	Resolved
5	Method takeUSDCRaised() transfer wrong funds	High	Resolved
6	Method _claim(...) calculates wrong shares and left values	High	Resolved
7	Centralization due to destroy() method	Medium	Resolved
8	selfDestruct Would Not Remove The ByteCode Of The Contract	Medium	Resolved
9	Missing disableInitializers Call in Proxy Upgradeable Contract Constructor	Medium	Resolved
10	Non-upgradeable contracts inherited	Low	Resolved
11	Missing validation for vesting points	Low	Resolved
12	Parameter privateTokenPrice is not validated	Low	Resolved
13	Missing validation for vesting Make Sure The Token In removeOtherERC20 Is Not The Token To Be Claimed	Low	Acknowledged
14	Previous staking contracts' STAKING role is not revoked	Low	Resolved
15	Unsafe cast	Low	Resolved
16	Flawed logic in _processPrivate() method	Low	Resolved
17	CEI pattern not followed	Low	Resolved

#	Title	Risk	Status
18	Vulnerability to frontrunning due to interplay between claim() and takeUSDCRaised()	Low	Resolved
19	Changeable length of vestingPoints leading to issues	Low	Resolved
20	Modifiers duplicated	Low	Resolved
21	Redundant check	Informational	Resolved
22	Unnecessary local variable in method usertaxRate()	Informational	Resolved
23	Remove console.log	Informational	Resolved
24	Unused imports	Informational	Resolved
25	Testnet Addresses Used	Informational	Partially Resolved
26	Floating Pragma Version in Solidity Files	Informational	Resolved
27	Lack of Event Emission in Privileged Functions	Informational	Resolved
28	Blacklisting is irreversible	Informational	Resolved
29	Unnecessary Safe Math is being utilized	Informational	Resolved

CRITICAL-1 | RESOLVED

Unrestricted call to destroyInstance

Locaton: Admin.sol
The `destroyInstance()` function within the smart contract poses a significant security risk due to its unrestricted accessibility. Any wallet or address can call this function, allowing any caller to delete the registered instance. This unrestricted access can have a severe impact on the ongoing operation of the smart contract, potentially leading to unauthorized destruction of critical contract components and disruption of essential functionalities.

```
function destroyInstance(
     address _instance
 ) external onlyExist(_instance) onlyIncoming(_instance) {
     _removeFromSales(_instance);
     ITokenSale(_instance).destroy();
 }
```

Recommendation:
Implement Access Control: Restrict access to the destroyInstance() function by implementing permission modifiers or access control mechanisms.

CRITICAL-2 | RESOLVED

Reentrance can lead to loss of funds

In Contract Vesting.sol, the method claim() allows users to claim the deposited tokens once the vesting time is over.

Since the deposited asset can be ETH/Native token as well, it will be sent to the user as per the following logic:

```
if (pctAmount != 0) {
            if (address(token) == address(1)) {
                (bool sent, ) = payable(sender).call{value: pctAmount}("");
                require(sent, "Failed to send BNB to receiver");
            } else {
                token.safeTransfer(sender, pctAmount);
            }
```

```
            s.index = uint128(i);
            s.amountClaimed += pctAmount;
        }
```

Here, Sending Ether is transferring the access of execution to the `sender` account, and then the `sender` account can call the `claim` method again. Since `s.index` is still not updated, the `pctAmount` will be calculated again and the same amount of ETH will be sent to the `sender`.

The sender can repeat this process until no ETH/native token is left in the contract. This way `sender` can steal all the ETH/native tokens from the contract.

This happens due to no reentrancy protection and not following the CEI (Check-Effects-Interaction) pattern.

Recommendation:
Update the `s.index` and `s.amountClaimed` before sending assets to the user. Also, check `s.amountClaimed` if necessary before the claim. Finally, add OZ Reentrancy guard as well.

HIGH-1 | RESOLVED

Incorrect Return Of userMaxAllc

In the contract TokenSale.sol the function calculateMaxAllocation calculate the max allocation for the user (L259) . The function incorrectly assign userMaxAlloc as the max allocation then userMaxAlloc > maxAllocation , in this case maxAllocation should have been assigned , therefore the allocation for the user would go beyond the max allowed allocation.

Recommendation:

Change the statement to `if (userMaxAllc > maxAllocation) {`

```
            return maxAllocation;
```

HIGH-2 | RESOLVED

Incorrect decimals precision

In Contract TokenSale.sol, the method _processPrivate(...) calculates the amount as below:

```
Staked storage s = stakes[_sender];
        uint256 amount = _amount * PCT_BASE;
        uint256 sum = s.amount + amount;
```

Here, amount = 1e6 * 1e18 = 1e24, if _amount is in USDC decimals 6.
Further same amount is transferred from user's wallet here:

```
usdc.safeTransferFrom(_sender, address(this), amount);
```

Since USDC has 6 decimals and amount is in 1e24, the amount transferred from user's wallet will be a lot more than it should be.
Similarly, userTaxAmount is in the same precision and transferred in the same way.

```
if (userTaxAmount > 0) {
            s.taxAmount += userTaxAmount;
            usdc.safeTransferFrom(_sender, marketingWallet, userTaxAmount);
}
```

Further state variables use the same decimal precision and the state is updated as follows:

```
s.amount += uint128(amount);
state.totalPrivateSold += uint128(amount);
```

Here s.amount and state.totalPrivateSold is being updated in 1e24 decimals.
If USDC decimals are 18, the situation will be much worse as now all values will be in 1e36.

Recommendation:
Update the _processPrivate method to correctly calculate the values in proper decimals.

Method takeUSDCRaised() transfer wrong funds

In Contract TokenSale.sol, the method takeUSDCRaised() calculates earned USDC as follows:

```
uint256 earned;

if (state.totalPrivateSold > state.totalSupplyInValue) {
    earned = uint256(state.totalSupplyInValue);
} else {
    earned = uint256(state.totalPrivateSold);
}
```

Here, state.totalPrivateSold is in 1e24 and state.totalSupplyInValue is in 1e18 so most likely `earned` will be state.totalSupplyInValue which will not be correct since real USDC balance can be less than totalSupplyInvalue.

Further, the following is checked if earned >= real USDC balance:

```
if (earned > 0) {
    uint256 bal = usdc.balanceOf(address(this));
    uint256 returnValue = earned <= bal ? earned : bal;
    usdc.safeTransfer(admin.wallet(), returnValue);
}
```

Here `earned` is in 1e18 but `bal` is in 1e6 since USDC has 6 decimals. So again, `returnValue` will be always `earned` which is 1e18 which is a lot more than what should be transferred to `admin.wallet`.

Recommendation:

Update the _processPrivate logic to ensure correct decimals for state.totalPrivateSold. Further, ensure that `earned` and `bal` are in the same decimal precision before comparing and transferring the amount to admin.

Method _claim(...) calculates wrong shares and left values

In Contract TokenSale.sol, the method _claim has the following logic:

```
  uint256 left;
```

```
        if (state.totalPrivateSold > (state.totalSupplyInValue)) {
uint256 rate = (state.totalSupplyInValue * PCT_BASE) /
                state.totalPrivateSold;
            _s.share = uint120((uint256(_s.amount) * rate) / PCT_BASE);
            left = uint256(_s.amount) - uint256(_s.share);
        } else {
            _s.share = uint120(_s.amount);          }

        return (_s.share, left);
```

Here, state.totalPrivateSold is in 1e24, and state.totalSupplyInValue is in 1e18 which will make the `if` condition mostly true leading to rate, share, and left value being calculated wrong.

Further, rate = 1e18 * 1e18 / 1e24 = 1e8
And share = 1e24 * 1e8 / 1e18 = 1e14
And left = 1e24 - 1e14.

Here all values are calculated in wrong decimals.

Recommendation:
Update the _processPrivate logic to ensure correct decimals for state.totalPrivateSold. Further calculate the rate, share, and left in correct decimals.

MEDIUM-1 | RESOLVED

Centralization due to destroy() method

In Contract TokenSale.sol, the method destroy() self-destructs the contracts and sends all funds to the admin wallet.

Since only the admin can call this method, the security of the funds in the contract is very much centralized.

Recommendation:
It is advised to decentralize the usage of these functions by using a multi-sig wallet with at least 2/3 or a 3/5 configuration of trusted users. Alternatively, a secure governance mechanism can be utilized for the same.

MEDIUM-2 | RESOLVED

selfDestruct Would Not Remove The ByteCode Of The Contract

The current instance of the TokenSale contract can be destroyed and the funds would be sent to the wallet address (L192 TokenSale.sol) , but the selfDestruct is deprecated now and the bytecode would still exist even after selfDestruct .
This can have varying effect on the users since they can still interact with the old instance (can be used for phishing purposes too).

Recommendation:
The risks for using selfDestruct should be acknowledged.

MEDIUM-3 | RESOLVED

Missing disableInitializers Call in Proxy Upgradeable Contract Constructor

Location: Vesting.sol, TokenSale.sol, Admin.sol

A concern arises due to the usage of a proxy upgradeable contract without calling disableInitializers in the constructor of the logic contract. This oversight introduces a severe risk, allowing potential attackers to initialize the implementation contract itself.

Recommendation:

Call `disableInitializers`: Include a call to `disableInitializers` in the constructor of the logic contract as recommended by OpenZeppelin here .

LOW-1 | RESOLVED

Non-upgradeable contracts inherited

Across the protocol, several contracts that are upgradeable are using the following import for AccessControl contract:

```
import "@openzeppelin/contracts/access/AccessControl.sol";
```

Instances:
Vesting.sol
Admin.sol

OpenZeppelin has provided an upgradeable version of the same contract and that needs to be used here.

The same goes for any other OpenZeppelin Contract to be imported in an upgradeable contract.

Recommendation:

Import the OZ's contracts as described here.

LOW-2 | RESOLVED

Missing validation for vesting points

In Contract Vesting.sol, the method initialize(...) sets the vesting points as follows:

```
(vestingPoints, ) = ascendingSort(_vestingPoints);
```

Here, vesting points are ordered as per the timestamp value but do not check if the time values are in the past.

Claim () can be done immediately if any past value is set.

Recommendation:
Validate that both points' timestamp value is >= block.timestamp.

LOW-3 | RESOLVED

Parameter privateTokenPrice is not validated

In Contract Admin.sol, the method _checkingParams has validation all the fields of the Struct param except privateTokenPrice.

It needs to be validated to make sure that it is an appropriate value and not `0`. Since this value is further used to calculate the total supply value for a Tokensale, it must not be `0` otherwise there would be unexpected results.

Recommendation:
Validate the parameter privateTokenPrice.

LOW-4 | ACKNOWLEDGED

Make Sure The Token In removeOtherERC20 Is Not The Token To Be Claimed

Users can claim their tokens using the claim() function in the Vesting.sol , there's also another functionality "removeOtherERC20" which rescues ERC20's accidentally sent to the contract . If the token address provided in the function is the address of the token to be claimed then the users won't be able to claim their rewards .

Recommendation:
Make sure the token address is not the token to be claimed.

LOW-5 | RESOLVED

Previous staking contracts' STAKING role is not revoked

In Contract Admin.sol, the method setStakingContract can set the new staking contract and give the STAKING role to that new contract.

In case, the staking contract is set/reset more than once, there would be multiple contracts with the STAKING role and this will allow all of those contracts to access methods that need to be accessed only by the current stakingContract as per logic.

Unsafe cast

In Contract TokenSale.sol, the method initialize calculated the totalSupplyInValue as follows:

```
state.totalSupplyInValue = uint128(
(uint256(_params.totalSupply) *
                uint256(_params.privateTokenPrice)) / 10 ** 18
// removes USDB hardcode of six, awful
        );
```

Here, `uint256` value is cast to `uint128` unsafely.

In _claim(..) method `s.share` is calculated as following:

```
_s.share = uint120((uint256(_s.amount) * rate) / PCT_BASE);
```

Here also `uint256` value is cast to `uint120` unsafely.

Recommendation:
Use OZ's SafeCast library.

LOW-7 | RESOLVED

Flawed logic in _processPrivate() method

In Contract TokenSale.sol, the method _processPrivate() has the following logic:

```
        if (sum > taxFreeAllcOfUser) { //@audit always true
            uint256 userTxRate = userTaxRate(sum, _sender);
            if (s.amount < taxFreeAllcOfUser) { //@audit true only once
                userTaxAmount = //@audit unreachable code after once
                    ((sum - taxFreeAllcOfUser) * userTxRate) /
                    POINT_BASE;
            } else {
                userTaxAmount = (amount * userTxRate) / POINT_BASE;
}
        }
```

Here, sum = s.amount + _amount, since _amount > 0 ⇒ sum > 0.
So the condition `sum > taxFreeAllOfUser` wil always be true as `taxFreeAllOfUser` is 0.
Similarly the condition `s.amount < taxFreeAllOfUser` can be true only once when s.amount == 0 but even in that case, userTaxAmount will be `amoiunt * userTxRate / POINT_BASE` only.

Recommendation:
Update the above logic with proper checks and conditions.

LOW-8 | RESOLVED

CEI pattern not followed

In Contract TokenSale.sol, the method claim() does not follow the CEI pattern as it transfers USDC to the users first (if s.taxAmount > 0) and then updates the s.claimed status. Here since it's USDC tokens, reentrancy is not possible but ensuring CEI pattern is advised.

Recommendation:
Update the logic to follow CEI pattern.

Vulnerability to frontrunning due to interplay between claim() and takeUSDCRaised()

Location: TokenSale.sol

The smart contract includes two distinct functions, `claim()` and `takeUSDCRaised()`, each serving specific purposes within the contract's functionality.

claim(): This function is designed to be callable by users, allowing them to claim certain benefits or rewards. Importantly, it is structured to be callable only once per user, ensuring fair distribution of benefits and preventing abuse or excessive claims.

takeUSDCRaised(): Conversely, `takeUSDCRaised()` is intended to be called only once in the smart contract lifespan, typically to distribute earnings or funds to an administrative wallet.

However, there exists a vulnerability arising from the interplay between these functions. When users call `claim()`, it affects their account balances within the contract. These balances, in turn, directly impact the amount that can be withdrawn or distributed when `takeUSDCRaised()` is subsequently called.

Vulnerability Scenario:

A potential attacker could exploit this relationship between `claim()` and `takeUSDCRaised()` to carry out frontrunning attacks. By repeatedly calling `claim()` to manipulate their account balances, the attacker can influence the amount available for withdrawal during the execution of `takeUSDCRaised()`. This manipulation can result in a negative impact on the intended distribution of earnings, potentially leading to financial losses or discrepancies.

Recommendation:
In order to ensure that `claim()` is called only after `takeUSDCRaised()` is once called, function `claim()` needs to check the variable `isRaiseClaimed`. This variable acts as a flag set to true once `takeUSDCRaised()` is called.

LOW-10 | RESOLVED

Changeable length of vestingPoints leading to issues

Location: Vesting.sol

Function `updateVestingPoints()` changes the array `vestingPoints`, while it is worth noting that caution is not taken on updating the value of this array. A new `vestingPoints` array with a different length can result in undesired behaviour. For instance assume the new `vestingPoints` is set to an array value with fewer elements (i.e. smaller length), the validation in function `claim() : require(s.index != vestingPoints.length, "already claimed");` is passed successfully despite the fact that this index should have been already claimed. New value of `vestingPoints` should be consistent with older value.

Recommendation:
Apply a constraint (i.e. `require` statement) to condition the new array value of `vestingPoints` in order to avoid inconsistentencies that might arise.

LOW-11 | RESOLVED

Modifiers duplicated

Location: Admin.sol

Functions `addOperator()` and `removeOperator()` carry out redundant checks. Modifier `onlyAdmin` checks whether the caller is `wallet` or not (i.e. owner of contract as referred to in `initialize`). Function `grantRole` as well carry out the same check: `onlyRole(getRoleAdmin(role))` which verifies that the role admin of `OPERATOR` is the caller of this function. The role admin of `OPERATOR` and `onlyAdmin` are both verifying that the caller is `wallet`.

Recommendation:
Apply a constraint (i.e. `require` statement) to condition the new array value of `vestingPoints` in order to avoid inconsistentencies that might arise.

```
 function addOperator(address _address) external virtual onlyAdmin {
        grantRole(OPERATOR, _address);
    }

    /**
     * @dev Only Admin can remove an Operator
     */

    function removeOperator(address _address) external virtual onlyAdmin
{
        revokeRole(OPERATOR, _address);
    }
```

This finding could have been more serious if there is a chance that role admin of `OPERATOR` be another account. As a result one modifier will contradict the other resulting in blocking the call of that function resulting in a denial of service.

Recommendation:
Use internal functions: `_grantRole` and `_revokeRole` in the implementations of `addOperator` and `removeOperator` instead.

INFORMATIONAL-1 | RESOLVED

Redundant check

In Contract TokenSale.sol, the method deposit checks the following:

```
require(_amount > 0, "TokenSale: 0 deposit");
```

And the method deposit calls `_processPrivate` which also checks:

```
require(_amount > 0, "TokenSale: Too small");
```

Since _processPrivate is called only by deposit() method, this check is redundant.

Recommendation:
Remove the redundant check.

INFORMATIONAL-2 | RESOLVED

Unnecessary local variable in method usertaxRate()

In Contract TokenSale.sol, the method userTaxRate() has the following logic:

```
uint256 userTaxFreeAllc = 0;

if (_amount > userTaxFreeAllc) { … }
```

This condition could simply be ` if (_amount > 0)` and there is no need of the variable usertaxFreeAllc.

Recommendation:
Remove the unnecessary local variable.

INFORMATIONAL-3 | RESOLVED

Remove console.log

In Contract TokenSale.sol, there is a console.log statement that should be removed before production deployment.

Recommendation:
Remove the unused logs.

INFORMATIONAL-4 | RESOLVED

Unused imports

In Contract Admin.sol, the following import is unused.

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

In TokenSale.sol, the following import is unused.

```
import "@chainlink/contracts/src/v0.8/interfaces/
AggregatorV3Interface.sol";
```

Recommendation:
remove the unused imports.

INFORMATIONAL-5 | RESOLVED

Testnet Addresses Used

Status - Partially Resolved

The USDC and wallet addresses used in TokenSale.sol (L88 and L89) are addresses meant for testnet and not for mainnet , these should be corrected before launch.

Recommendation:
Change the addresses to mainnet addresses.
Fix - Issue is unaddressed and remains unresolved (commit 65e4d1c).
Fix - Commit - fixed for the address of usdc. marketingWallet still assigned `0×6507fFd283c32386B6065EA89744Ade21515e91E` which does not exist in the network.

INFORMATIONAL-6 | RESOLVED

Floating Pragma Version in Solidity Files

Location: All
The Solidity files in this codebase contain pragma statements specifying the compiler version in a floating manner (e.g., `^0.8.0`, `^0.8.4`, `^0.8.11`). 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 automatic adoption of potentially incompatible compiler versions.

INFORMATIONAL-7 | RESOLVED

Lack of Event Emission in Privileged Functions

Location: Vesting.sol, VestingFactory.sol, TokenSale.sol, Admin.sol

During the security audit of the smart contract, an issue has been identified in the implementation of privileged functions, one example is function `setOperator(address)`. 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 - Finding is addressed in Admin.sol only in commit 65e4d1c. While in the rest of the contracts mentioned, most privileged functions are called without emitting events.

Fix - Fixed in `e5cb23a`.

INFORMATIONAL-8 | RESOLVED

Blacklisting is irreversible

Location: Admin.sol

In function `addToBlacklist`, addresses are blacklisted without any chance to go back from it. Once function is called against a given address even by mistake the admin can not remove that item from blacklist.

Recommendation:
It is a sound suggestion to implement a function that removes address from blacklist.

Unnecessary Safe Math is being utilized

Location: Vesting.sol, TokenSale.sol, Admin.sol

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.

One example in Vesting.sol:

```
for (uint256 i = 0; i < _users.length; i++) {
     userdetails[_users[i]].userDeposit = _amount[i];
     amount += _amount[i];
 }
```

Preferrable to implement it as such:

```
for (uint256 i = 0; i < _users.length;) {
     userdetails[_users[i]].userDeposit = _amount[i];
     amount += _amount[i];
     unchecked {
         i++;
     }
 }
```

Recommendation:
Utilize the use of `unchecked` whenever possible as long as arithmetic safety is ensured within the context of calculation.

	Vesting VestingFactory Admin TokenSale ERC20 Factory SimpleERC20
Reentrance	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 ZAP 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 ZAP team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo