ZAP.

SMART CONTRACTS REVIEW

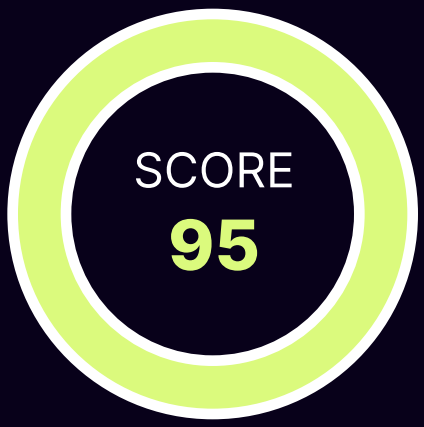
zokyo

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
- 2 High issues: 2 resolved = 0 points deducted
- 4 Medium issues: 3 resolved and 1 acknowledged = - 5 points deducted
- 2 Low issues: 2 resolved = 0 points deducted
- 5 Informational issues: 5 resolved = 0 points deducted

Thus, 100 - 5 = 95

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

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

Last commit: 6f731f14a8d1d86476af42f90e6faf8df35e22ad

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

- Admin
- TokenSale

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 setConfig	Critical	Resolved
2	Incorrect Claim Logic	Critical	Resolved
3	Incorrect Decimals For unsoldTokens	High	Resolved
4	Vesting Logic Would Keep Reverting For An Edge Case	High	Resolved
5	Lack of ERC20 Token Transfer Validation	Medium	Resolved
6	Missing disableInitializers Call in Proxy Upgradeable Contract Constructor	Medium	Resolved
7	Incorrect Calculation Of amountFinal	Medium	Resolved
8	0 Slippage Protection	Medium	Acknowledged
9	Reentrance Attack Exploit in claim() Function	Low	Resolved
10	Non-upgradeable contracts inherited	Low	Resolved
11	The address of USDB refers to a test chain	Informational	Resolved
12	Lack of Event Emission in Privileged Functions	Informational	Resolved
13	Redundant Checks	Informational	Resolved
14	Confusing Function Name And Comments	Informational	Resolved
15	Improving Clarity and Functionality of setConfig Method	Informational	Resolved

Unrestricted call to setConfig

Location: TokenSaleUSDB.sol, TokenSaleETH.sol

The `setConfig()` function within the smart contracts 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. The validation in this function `require(creator == _creator,"Only Creator can Call");` checks that the `_creator` is the registered creator recorded in the contract state. But this check does not hinder the attacker because the attacker can simply call this function with the intended `_creator = creator` in order to carry out this call successfully.

```
function setConfig(Config memory _config, address _creator) external {
      require(creator == _creator,"Only Creator can Call");
      config = _config;
  }
```

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. Attacker is capable of changing the `config` without the consent of the creator.

Recommendation:

Validate that `msg.sender` is the registered creator.

CRITICAL-2 | RESOLVED

Incorrect Claim Logic

Users can claim the extra USDB (same issue for TokenSaleETH.sol) using the function claim in TokenSaleUSDB.sol , if s.left > 0 that would mean there are USDB to claim , s.left is compared with the current balance of USDB in the contract at L263 to calculate “left” , these left tokens are then transferred to the user.
But the s.left updated at L265 is incorrect , instead of deducting left from s.left , the whole of s.left is deducted and emptied out , therefore in a condition where the contract had insufficient USDB to fulfill the claim the s.left would be emptied anyhow and the user won’t be able to claim the rest .

Recommendation:
Change the update to s.left -= left

HIGH-1 | RESOLVED

Incorrect Decimals For unsoldTokens

In the contract TokenSaleETH (same issue in TokenSaleUSDB) , in the function takeUSDBRaised() the unsoldTokens are calculated as (unsold * (10 ** ETH_DECIMAL))/ tokenPrice , this amount is the amount of ‘tokenAddress’ tokens therefore should be in tokenAddress decimals , but the above formula always returns in 1e18 rather than tokenAddress decimals . Therefore if say the tokenAddress is a token with 12 decimals then the unsoldTokens would still be in 1e18 which would be a value 1e6 times higher than expected.

Recommendation:
Normalise the decimals accordingly (a division by decimal difference).

Vesting Logic Would Keep Reverting For An Edge Case

The normal flow in the system is , users deposit , they deposit is raised and liquidity is added to pancakeSwap and users can vest their shares using the function vesting() .
In the contract TokenSaleUSDB (same for TokenSaleETH.sol) the decimalDifference is calculated at L281 which is used to normalise the amountFinal . But if the tokenAddress's decimal is more than 18 then the calculation at L281 would always revert due to overflow.

Therefore a malicious user can create a TokenSale where token is a token with 22 decimals and make the users deposit their share into the pool , but the users would never be able to claim their vest due to the overflow error .

Recommendation:
Restrict token to 18 decimals or normalise accordingly.

Lack of ERC20 Token Transfer Validation

Location: Admin.sol, TokenSaleUSDB.sol

Function `createPoolNew()` initiates ERC20 token transfers without validating the return value of the transfer operation. This oversight can result in unhandled transfer failures, compromising the contract's functionality and exposing it to potential vulnerabilities.

```
310     IERC20D(_params.tokenAddress).transferFrom(msg.sender, instance,
(_params.totalSupply + _params.tokenLiquidity));
```

There are also two occurrences in TokenSaleUSDB.sol in lines 236, 246:

```
236     if(params.tokenLiquidity - tokenAmount > 0)
IERC20D(params.tokenAddress).transfer(creator, params.tokenLiquidity -
tokenAmount);
        ...
246     if (liquidity > 0) IERC20D(pair).transfer(creator,liquidity);
```

Risks: Unintended Consequences: Failing to validate token transfers may lead to unhandled failures, causing unintended consequences in contract state and potentially resulting in financial losses.

Recommendation:
To ensure secure and reliable ERC20 token transfers, it is recommended to use SafeERC20 Library to ensure safe and reliable token transfers, reducing the risk of vulnerabilities associated with ERC20 token handling.

Fix - Issue is addressed in commit `021212d` and Fixed.

MEDIUM-2 | RESOLVED

Missing disableInitializers Call in Proxy Upgradeable Contract Constructor

Location: Admin.sol, TokenSaleUSDB.sol, TokenSaleETH.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.

MEDIUM-3 | RESOLVED

Incorrect Calculation Of amountFinal

In contract TokenSaleETH.sol (same for TokenSaleUSDB.sol) when vesting() is called it calculates the amountFinal value at L261 . When there is no decimal difference and the calculated amount is less than balance then that calculated amount is assigned as amountFinal , else the balance is the amountFinal.
If there is a decimal difference and the calculated amount is more than balance then that amount is assigned as the amountfinal which is incorrect , we should assign bal in this case as amountFinal , the code is assigning a higher value than the available balance to the amountfinal.

Recommendation:

Change the code as suggested.

MEDIUM-4 | ACKNOWLEDGED

0 Slippage Protection

When Liquidity is added to the pancake router , the amountAMin and amountBMin params are being passed as `0` (L231 and L232 in TokenSaleUSDB , same for TokenSaleETH) which essentially means `0` protection against slippage impacts , these values should be assigned to a non-zero safe value to ensure the slippage is within accepted range.

Recommendation:

Assign the values to non-zero values.

LOW-1 | RESOLVED

Reentrance Attack Exploit in claim() Function

Location: TokenSaleETH.sol, TokenSaleUSDB.sol

The `claim()` function is susceptible to a reentrancy attack due to its design allowing external calls (i.e. line 240) to be made to other contracts (i.e. msg.sender) before the completion of its own state changes and balance updates. An attacker can exploit this vulnerability by invoking the `claim()` function and subsequently reentering it multiple times, potentially draining the contract's funds by manipulating its state or performing unauthorized transfers.

```
225 function claim() external {
226     require(isRaiseClaimed, "takeUSDBRaised not called");
227     require(
228         uint8(epoch) > 1 && !admin.blockClaim(address(this)),
229         "Not time or not allowed"
230     );
231
232     Staked storage s = stakes[msg.sender];
233     require(s.amount != 0, "No Deposit");
234     if(s.share == 0) (s.share, s.left) = _claim(s);
235     require(s.left != 0, "Nothing to Claim");
```

```
237     if(s.left > 0){
238         uint bal = address(this).balance;
239         uint left = s.left < bal ? s.left : bal;
240         ETHTransfer(msg.sender, left);
241         s.left -= s.left;
242         emit Claim(msg.sender, left);
        }
    }
```

Recommendation:

Use Reentrancy guard: Implement reentrancy guard or state variables to prevent reentrancy by blocking reentrant calls to the claim() function while a transaction is in progress.

Fix - Issue is addressed in commit 021212d and partially resolved. Reentrancy guard is added to the contract and nonReentrant modifier is being applied to the function. The issue now is effectively mitigated but it is worth noting that ReentrancyGuard being used is not an upgradeable version.

It is also recommended that s.left -= left; be executed before ETHTransfer(msg.sender, left); since using checks-effects-interactions pattern helps mitigate that exploitation.

Fix - Issue revisited in commit 1474b46 in which client moved s.left -= left before ETHTransfer to abide by CEI pattern this in turn lowers the severity of reentrancy.

Fix: At commit 5c0e17cb92 : Both TokenSaleETH.sol and TokenSaleUSDB.sol inherit the Upgradeable version of ReentrancyGuard as suggested , however it is not initialized to the proper state in Function initialize.

Non-upgradeable contracts inherited

Location: Admin.sol, TokenSaleETH.sol, TokenSaleUSDB.sol

Across the protocol, several contracts that are upgradeable are inheriting non upgradeable Base Contracts:

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

OpenZeppelin has provided an upgradeable version of the same contracts 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.

Fix: At commit 5c0e17cb92 : Client inherited the required upgradeable contracts and initialized them in the initialize function in Admin.sol. In both TokenSaleETH.sol and TokenSaleUSDB.sol the Upgradeable version of ReentrancyGuard is inherited as suggested , however it is not initialized to the proper state in Function initialize.

INFORMATIONAL-1 | RESOLVED

The address of USDB refers to a test chain

Location: Admin.sol

In function `initialize()` it is noted that contract not ready to operate properly because address of USDB refers to the testnet while it should be `0x4300000000000000000000000000000000000003`.

```
57  function initialize(address _owner) public initializer {
58      _setupRole(DEFAULT_ADMIN_ROLE, _owner);
59      _setRoleAdmin(OPERATOR, DEFAULT_ADMIN_ROLE);
60      wallet = _owner;
61      platformFee = 2 * (10 ** 17);
62      platformTax = 500;
63      USDB = 0xA9F81589Cc48Ff000166Bf03B3804A0d8Cec8114;
    }
```

INFORMATIONAL-2 | RESOLVED

Lack of Event Emission in Privileged Functions

Location: Admin.sol, TokenSaleUSDB.sol, TokenSaleETH.sol

During the security audit of the smart contract, an issue has been identified in the implementation of privileged functions, one example is function `setMasterContractUSDB(addres )`. 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 - Issue has been addressed in commit `021212d` and partially resolved. The issue in contracts TokenSaleUSDB and TokenSaleETH is not being addressed. Functions like setUSDB and setConfig are admin functions that are being invoked making significant privileged changes without announcing these changes.

INFORMATIONAL-3 | RESOLVED

Redundant Checks

The check at L261 (TokenSaleUSDB.sol) is redundant , this is because the require statement at L259 ensures s.left is more than `0`.
Another instance is check at L181 TokenSale.sol , this check would always be true due to the require statement at L174.

Recommendation:
The if condition can be removed.

INFORMATIONAL-4 | RESOLVED

Confusing Function Name And Comments

The function takeUSDBRaised() in the contract TokenSaleETH.sol does not take the USDB raised in the epoch rather it's the ETH that was deposited in the round , it should be named accordingly . Plus , all the comments in the contract has been copied from the TokenSaleUSDB contract and should be changed.

Recommendation:
Change the comments and the function name accordingly.

Improving Clarity and Functionality of setConfig Method

Location: TokenSaleUSDB.sol, TokenSaleETH.sol

The `setConfig` function is unclear and could lead to confusion. The function appears to intend that only the creator of the contract should call it. However, the actual caller in the context of the protocol is the Admin contract via the `createPoolNew` function. This discrepancy may cause misunderstanding for developers attempting to interact with the contract.

Furthermore, the function is designed to be called only once, and it could be more intuitive to call it within the `initialize` function rather than as a separate transaction.

Recommendation:
Consider revising the function to enhance clarity and align its behavior with the intended usage of the contract. Additionally, ensure that comments or documentation are provided to clarify the intended usage and the role of the caller. This will help improve readability and reduce the risk of misinterpretation by developers interacting with the contract.

	Admin TokenSale
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