Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: OZONE
Website: ozonex.tech
Platform: BNB Smart Chain (BSC)
Language: Solidity
Date: November 6th, 2025

Table of contents

Email: audit@EtherAuthority.io

THIS IS A SECURITY AUDIT REPORT DOCUMENT AND MAY CONTAIN INFORMATION THAT IS CONFIDENTIAL. THIS INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES THAT CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Email: audit@EtherAuthority.io

Introduction

EtherAuthority was contacted by the OZONE team to perform a security audit of the OZONE smart contract's code. The audit was conducted using manual analysis and automated software tools. This report presents all the findings regarding the audit performed on November 6th, 2025.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

OzoneXStaking is an advanced DeFi staking contract that allows users to stake **OZONE tokens** and earn rewards in **two different forms** — **USDT** or **OZONE** — depending on their preference and the pool's configuration.

It supports **multiple staking pools**, each with its own rules for APY, minimum/maximum stake limits, reward intervals, and automatic burn features.

Key Features:

1. **Dual Reward Options**
    - Users can earn rewards in **USDT** or **OZONE**.
    - Each pool can allow or restrict OZONE rewards.
2. **Multi-Pool System**
    - Admins can create multiple pools with different parameters (APY, claim interval, etc.).
    - Each pool is independent and customizable.
3. **Automatic Token Burn**
    - When a user's rewards reach the maximum cap (e.g., 300% of the staked amount), their tokens can be **auto-burned** for supply control and sustainability.
4. **Flexible Claiming**
    - Users can claim rewards after a set interval (default: 15 days).
    - They can select the reward type when claiming.

5. **Manual Unstake**
    - Users can unstake at any time before reaching the maximum reward limit and withdraw their staked tokens.
6. **Reserve-Based Rewards**
    - Rewards are paid from **USDT and OZONE reserves, which are** funded by the contract owner.
    - Prevents reward generation without actual liquidity backing.
7. **Admin Control**
    - Create, update, or deactivate pools.
    - Fund or withdraw reward reserves.
    - Update the OZONE price for conversion.
    - Pause/unpause the entire system in emergencies.
8. **Proof of Reserves**
    - Integrates with the OZONE token contract to display on-chain reserve data for transparency.

.Audit scope

Name	**Code Review and Security Analysis Report for OZONE Smart Contract**
Website	ozonex.tech
Staking Apps	ozonehub.io
Platform	**BNB Smart Chain (BSC)**
Language	**Solidity**
File	OzoneXStaking.sol
Initial Code Link	0x9316865b229045dca5ab5058059ca84a9fe23aa9
Audit Date	November 6th, 2025
Revised Audit Date	November 7th, 2025

Claimed Smart Contract Features

Claimed Feature Details	Our Observation

File: OzoneXStaking.sol
Token Distribution:
Tokenomics:
• **Total Supply:** 1 billion $OZONE tokens
 	
Token Utility Framework:
 	
Core Staking Features:
• Multi-pool staking system (each pool has unique APY, limits, and intervals).
• Supports staking of **OZONE tokens**.
• Automatic calculation of **daily rewards** based on pool APY.
• Configurable **claim interval** (default: 15 days).
• Maximum reward cap (default: **300%** of staked amount).
 	
Dual Reward System:
• Users can **choose** to receive rewards in:
  ○ **USDT** (stable reward)
  ○ **OZONE** (native token)
• Each pool can **enable/disable** OZONE rewards individually.
• Dynamic conversion based on real-time ozone price.
 	
Reward Claim & Management:
• Claim rewards anytime after the claim interval.
• Supports **manual or auto-claim** during unstake.
• Tracks:
  ○ Total rewards claimed.
	YES, this is valid. The smart contract owner controls these functions, so the owner must handle the private key of the owner's wallet very securely.
	Because if the private key is compromised, then it will create problems.

○ Rewards claimed in USDT and OZONE separately. ● Automatically checks the **reserve balance** before paying rewards. **<u>Auto Burn Mechanism:</u>** ● If total rewards reach the maximum cap (e.g., 300%), → Staked tokens are **automatically burned** to a dead address. ● Reduces supply and maintains token value. ● Emits a detailed burn event for transparency. **<u>Pool Management (Admin)</u>**: ● **Create**, **update**, or **deactivate** staking pools. ● Set custom: ○ APY, claim interval, min/max stake ○ Max reward %, auto-burn flag, and reward type allowance ● Fully modular — multiple active pools supported. **<u>Reserves & Funding:</u>** ● Separate reserves for USDT and OZONE rewards. ● **The owner can:** ○ Fund or withdraw reserves. ○ Track total distributed rewards and reserve balances. ● Ensures proof-of-reserve-backed payouts. **<u>Price & Conversion Control:</u>** ● The owner can manually set the **OZONE price** (1 USDT = N OZONE). ● Used for OZONE reward calculation.	

• Designed for future integration with on-chain oracles.
 	
<u>Security & Safety:</u>
• **ReentrancyGuard** — prevents double claim attacks.
• **Ownable** — admin-only functions protected.
• **Pausable** — emergency pause for all operations.
• **Emergency Withdraw** — owner can recover tokens when paused.
 	
<u>Transparency & Tracking:</u>
• Complete pool & user stake history accessible via view functions.
• On-chain events for all actions:
  ○ Pool creation/update
  ○ Stake, claim, unstake, burn
  ○ Reserve funding & withdrawal
• Integrates with the **OZONE Proof-of-Reserves** contract.	

Audit Summary

According to the standard audit assessment, the Customer`s solidity-based smart contract is **“Secured”**. Also, these contracts contain owner control, which does not make them fully decentralized.

We utilized various tools, including Slither, Solhint, and Remix IDE. This finding is also based on a critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed, and applicable vulnerabilities are presented in the Audit overview section. A general overview is presented in the "AS-IS" section, and all identified issues are listed in the "Audit Overview" section.

We found 0 critical, 1 high, 1 medium, 0 low, and 2 very low-level issues.

We confirm that all issues are acknowledged.

Investor Advice: A technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner-controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	The solidity version is not specified	Passed
	The solidity version is too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack a check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks an event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Moderated
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Moderated
Gas Optimization	“Out of Gas” Issue	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage is not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Code Quality

This audit scope has 1 smart contract file. A smart contract contains Libraries, Smart contracts, inheritance, and Interfaces. This is a compact and well-written smart contract.

The libraries in OZONE are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address, and its properties/methods can be reused many times by another contract in OZONE.

The OZONE team has not provided scenarios and unit test scripts, which would have helped to determine the integrity of the code automatically.

The code sections are **well-**commented in the smart contract. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given an OZONE smart contract code in the form of a bscscan.com link. The smart contract link is mentioned in the table above.
As mentioned above, the code parts are **well commented**, and the logic is straightforward. Thus, it is easy to understand the programming flow and complex code logic quickly. Comments are constructive in understanding the overall architecture of the protocol.

Another source of information was its official Website: https://ozonex.tech and Whitepaper: https://drive.google.com/file/d/1BnhideVxbn2RjL52P_fHJNFlMvbhC7TX/view, which provided rich information about the project architecture.

Use of Dependencies

According to our observation, the libraries utilized in this smart contract infrastructure are based on well-known industry-standard, open-source projects.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

OzoneXStaking.sol

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	createPool	external	access only Owner	No Issue
3	updatePool	external	access only Owner	No Issue
4	deactivatePool	external	access only Owner	No Issue
5	stake	external	Passed	No Issue
6	calculateAvailableRewards	read	Passed	No Issue
7	calculateRewardDistribution	read	Passed	No Issue
8	canClaim	read	Passed	No Issue
9	claimRewards	external	Passed	No Issue
10	claimRewardsWithType	external	Passed	No Issue
11	_claimRewards	internal	Passed	No Issue
12	_autoBurnTokens	write	Passed	No Issue
13	unstake	external	Lock period not enforced properly	Acknowledged
14	fundUSDTReserves	external	access only Owner	No Issue
15	fundOzoneReserves	external	access only Owner	No Issue
16	withdrawUSDTReserves	external	access only Owner	No Issue
17	withdrawOzoneReserves	external	access only Owner	No Issue
18	setOzonePrice	external	access only Owner	No Issue
19	getPool	external	Passed	No Issue
20	getUserStake	external	Passed	No Issue
21	getUserStakeCount	external	Passed	No Issue
22	getRewardBreakdown	external	Passed	No Issue
23	getStakingStats	external	Passed	No Issue
24	getOZONEProofOfReserves	external	Variable Shadowing Warning	Acknowledged
25	getVersion	external	Passed	No Issue
26	getTotalActiveStakes	external	Passed	No Issue
27	pause	external	access only Owner	No Issue
28	unpause	external	access only Owner	No Issue
29	emergencyWithdrawUSDT	external	Emergency withdrawals don't update reserves	Acknowledged
30	emergencyWithdrawOZONE	external	Emergency withdrawals don't update reserves	Acknowledged
31	nonReentrant	modifier	Passed	No Issue
32	_nonReentrantBefore	write	Passed	No Issue
33	_nonReentrantAfter	write	Passed	No Issue
34	_reentrancyGuardEntered	internal	Passed	No Issue
35	onlyOwner	modifier	Passed	No Issue

36	owner	read	Passed	No Issue
37	_checkOwner	internal	Passed	No Issue
38	renounceOwnership	write	access only Owner	No Issue
39	transferOwnership	write	access only Owner	No Issue
40	_transferOwnership	internal	Passed	No Issue
41	whenNotPaused	modifier	Passed	No Issue
42	whenPaused	modifier	Passed	No Issue
43	paused	read	Passed	No Issue
44	_requireNotPaused	internal	Passed	No Issue
45	_requirePaused	internal	Passed	No Issue
46	_pause	internal	Passed	No Issue
47	_unpause	internal	Passed	No Issue

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss, etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial.
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc., code snippets that can't have a significant impact on execution.
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No critical-severity vulnerabilities were found

High Severity

(1) Lock period not enforced properly:

```
/**
 * @dev Manual unstake before reaching 300%
 */
function unstake(uint256 _stakeIndex) external nonReentrant whenNotPaused {    infinite gas
    require(_stakeIndex < userStakes[msg.sender].length, "Invalid stake index");

    UserStake storage userStake = userStakes[msg.sender][_stakeIndex];
    require(userStake.isActive && !userStake.isBurned, "Stake is not active");

    // Claim pending rewards if possible
    if (canClaim(msg.sender, _stakeIndex)) {
        (uint256 pendingRewards,) = calculateAvailableRewards(msg.sender, _stakeIndex);
        if (pendingRewards > 0) {
            _claimRewards(_stakeIndex, userStake.preferredRewardType);
        }
    }

    uint256 unstakeAmount = userStake.amount;

    // Mark as inactive
    userStake.isActive = false;

    // Update stats
    pools[userStake.poolId].totalStaked -= unstakeAmount;
    activeStakeCount--;

    // Return staked tokens
    require(ozoneToken.transfer(msg.sender, unstakeAmount), "Transfer failed");

    emit UserUnstaked(msg.sender, _stakeIndex, unstakeAmount);
}
```

The Staking system design implies locked staking, but users can withdraw anytime.
This breaks the expected staking commitment.

Resolution: Implement a lock period per pool and block withdrawals prior to maturity. Allow exit only after the lock ends or apply a penalty burn/fee.

Status: Acknowledged

Comments: Client will make sure that the unstake will not be executed from the UI.

Medium

(1) Emergency withdrawals don't update reserves:

```
/**
 * @dev Emergency withdraw (only when paused)
 */
function emergencyWithdrawUSDT(uint256 _amount) external onlyOwner whenPaused {    infinite gas
    require(usdtToken.transfer(owner(), _amount), "Transfer failed");
}

/**
 * @dev Emergency withdraw OZONE (only when paused)
 */
function emergencyWithdrawOZONE(uint256 _amount) external onlyOwner whenPaused {    infinite gas
    require(ozoneToken.transfer(owner(), _amount), "Transfer failed");
}
```

The emergencyWithdrawUSDT() and emergencyWithdrawOZONE() functions transfer tokens directly to the contract owner but do not update the corresponding internal reserve variables (stakingUSDTReserves and ozoneReserves).
This creates a mismatch between the actual token balances and the contract's internal accounting.

For example:
If the contract holds 500 OZONE tokens and ozoneReserves records 200, and the owner performs an emergency withdrawal of 400 OZONE, the contract will now hold only 100 tokens — but ozoneReserves will still incorrectly indicate 200.
As a result, subsequent reward claims or calculations that rely on reserve balances may revert or produce incorrect values because the reserves no longer match actual liquidity.

Resolution: Subtract the withdrawn amount from the corresponding reserve variable (stakingUSDTReserves or ozoneReserves) whenever an emergency withdrawal is executed.

Add validation to ensure that the owner cannot withdraw more than the recorded reserve balance.

Optionally, include a syncReserves() function that allows the contract to realign its internal reserve variables with on-chain token balances after any manual intervention.

Status: Acknowledged

Comments: The owner will take care of this.

Low

No low-severity vulnerabilities were found

Very Low / Informational / Best practices:

(1) Unused variables:

```
// Constants - Production Configuration
uint256 public constant MAX_REWARD_PERCENTAGE = 30000; // 300% maximum
uint256 public constant DEFAULT_CLAIM_INTERVAL = 1296000; // 15 days in seconds
uint256 public constant MONTH_DURATION = 2592000; // 30 days in seconds
```

There is a MAX_REWARD_PERCENTAGE variable defined, but not used anywhere.

Resolution: Remove unused variables from the code.

Status: Acknowledged

(2) Variable Shadowing Warning:

```
Warning: This declaration shadows an existing declaration.
--> OzoneXStaking/OzoneXStaking.sol:1055:63:
|
1055 | function getOZONEProofOfReserves() external view returns (uint256 ozoneReserves, uint256 usdtReserves) {
| ^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
--> OzoneXStaking/OzoneXStaking.sol:510:5:
|
510 | uint256 public ozoneReserves; // OZONE reserves untuk dual rewards
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Warning: This declaration shadows an existing declaration.

A function parameter or local variable uses the same name as a state variable. This can cause unexpected behavior or confusion in the contract.

Resolution: Rename either the state variable or the function return variable to avoid the name conflict.

For example, use ozoneReservesLocal inside the function instead of ozoneReserves.

Status: Acknowledged

Centralization

This smart contract has some functions that can be executed by the Admin (Owner) only. If the admin wallet private key were compromised, then it would create trouble. The following are Admin functions:

OzoneXStaking.sol

- **createPool:** Allows the owner to create a new staking pool with configurable APY, limits, intervals, and dual reward options.
- **updatePool:** Updates parameters of an existing pool (APY, limits, burn rules, reward type) by the owner.
- **deactivatePool:** Deactivates a pool, preventing new stakes while keeping existing stakes intact by the owner.
- **fundUSDTReserves:** Allows the owner to add USDT to the contract for future staking rewards.
- **fundOzoneReserves:** Allows the owner to add OZONE to the reward reserve pool.
- **withdrawUSDTReserves:** Allows the owner to withdraw available USDT reserves from the contract.
- **withdrawOzoneReserves:** Allows the owner to withdraw available OZONE reserves from the contract.
- **setOzonePrice:** Updates the OZONE token's price used for USDT↔OZONE reward conversions by the owner.
- **pause:** Pauses all staking, claiming, and unstaking operations by the owner.
- **unpause:** Resumes contract operations after being paused by the owner.
- **emergencyWithdrawUSDT:** Allows the owner to withdraw USDT during an emergency when paused.
- **emergencyWithdrawOZONE:** Allows the owner to withdraw OZONE during an emergency when paused.

Conclusion

We were given a contract code in the form of a bscscan.com weblink. We have used all possible tests based on the objects in the given file. During our analysis, we identified 1 high, 1 medium, and 2 informational severity issues in the smart contract. All identified problems have been acknowledged. **Therefore, we confirm that the smart contract has been successfully reviewed and deployed on the mainnet.**

Since possible test cases can be unlimited for such a smart contract protocol, we provide no such guarantee of future outcomes. We have utilized the latest static analysis tools and manual observations to cover as many test cases as possible, ensuring comprehensive scanning of all relevant areas.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed smart contract, based on the standard audit procedure scope, is **"Secured".**

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of the systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and white box penetration testing. We look at the project's website to get a high-level understanding of the functionality of the software under review. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early, even if they are later shown not to represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation are an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

The EtherAuthority team has analyzed this smart contract by the best industry practices as of the date of this report, about cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

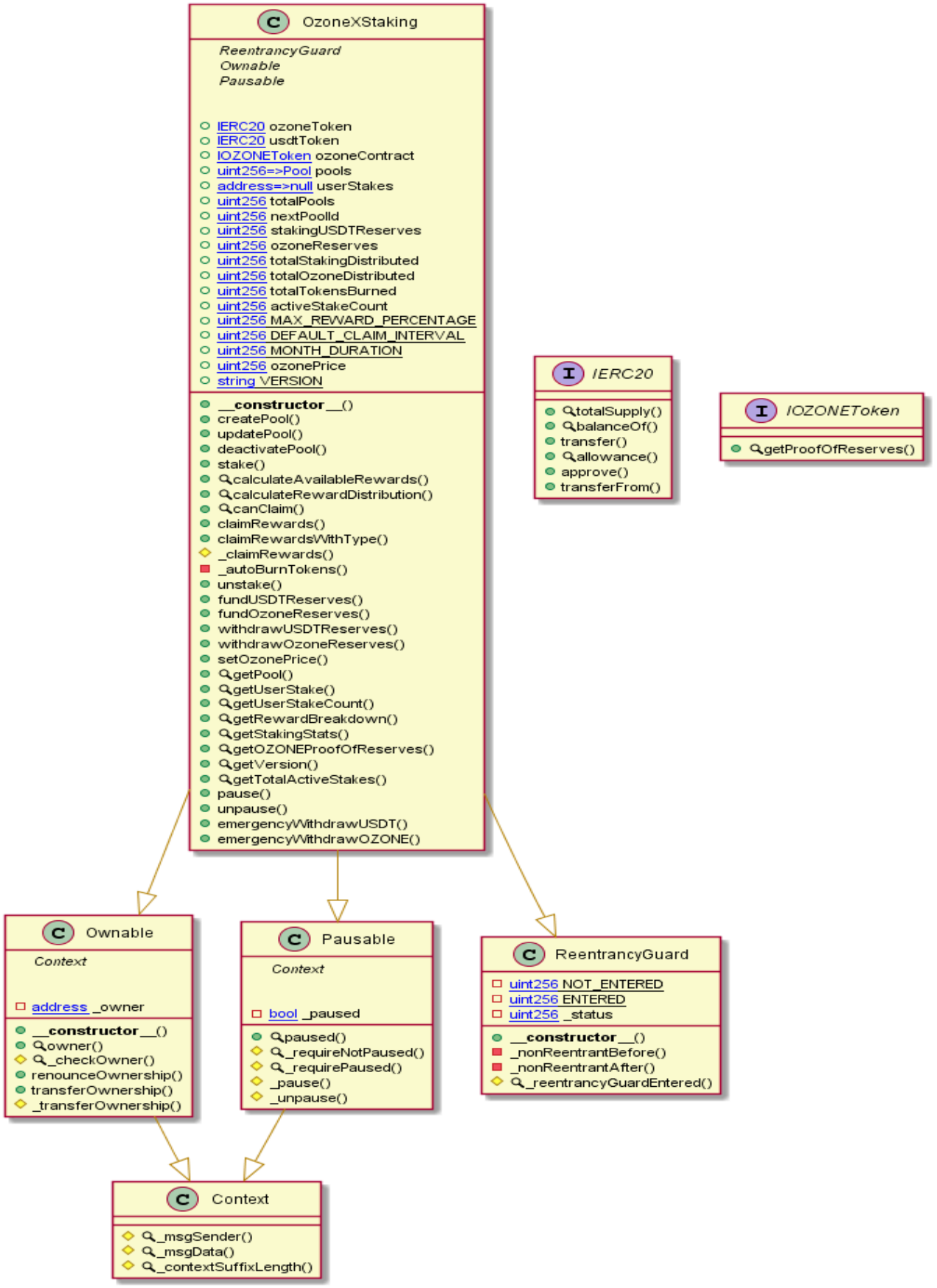
Because the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best to conduct the analysis and produce this report, it is important to note that you should not rely on this report alone. We also suggest conducting a bug bounty program to confirm this smart contract's high security level.

Technical Disclaimer

A smart contract is deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee the explicit security of the audited smart contract.

Appendix

Code Flow Diagram - OZONE



Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and quickly prototype custom analyses. The study includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We analyzed the project together. Below are the results.

Slither Log >> OzoneXStaking.sol

```
INFO:Detectors:
OzoneXStaking.calculateAvailableRewards(address,uint256) (OzoneXStaking.sol#722-756) performs a multiplication on the result of a division:
	- daysElapsed = timeElapsed / 86400 (OzoneXStaking.sol#733)
	- dailyReward = (userStake.originalAmount * pool.monthlyAPY) / 10000 / 30 (OzoneXStaking.sol#738)
	- totalReward = dailyReward * daysElapsed (OzoneXStaking.sol#739)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
OzoneXStaking.calculateAvailableRewards(address,uint256) (OzoneXStaking.sol#722-756) uses a dangerous strict equality:
	- daysElapsed == 0 (OzoneXStaking.sol#735)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in OzoneXStaking._claimRewards(uint256,OzoneXStaking.RewardType) (OzoneXStaking.sol#810-862):
	External calls:
	- require(bool,string)(usdtToken.transfer(msg.sender,usdtAmount),USDT transfer failed) (OzoneXStaking.sol#844)
	State variables written after the call(s):
	- ozoneReserves -= ozoneAmount (OzoneXStaking.sol#848)
	OzoneXStaking.ozoneReserves (OzoneXStaking.sol#514) can be used in cross-function reentrancies:
	- OzoneXStaking.fundOzoneReserves(uint256) (OzoneXStaking.sol#941-945)
	- OzoneXStaking.getStakingStats() (OzoneXStaking.sol#1031-1054)
	- OzoneXStaking.ozoneReserves (OzoneXStaking.sol#514)
	- OzoneXStaking.withdrawOzoneReserves(uint256) (OzoneXStaking.sol#960-965)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
OzoneXStaking.getOZONEProofOfReserves() (OzoneXStaking.sol#1059-1061) ignores the return value by ozoneContract.getProofOfReserves() (OzoneXStaking.sol#1060)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Reentrancy in OzoneXStaking.fundUSDTReserves(uint256) (OzoneXStaking.sol#932-936):
	External calls:
	- require(bool,string)(usdtToken.transferFrom(msg.sender,address(this),_amount),USDT transfer failed) (OzoneXStaking.sol#933)
```

```
        State variables written after the call(s):
        - stakingUSDTReserves += _amount (OzoneXStaking.sol#934)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
OzoneXStaking.getUserStake(address,uint256) (OzoneXStaking.sol#991-994) uses timestamp for
comparisons
        Dangerous comparisons:
        - require(bool,string)(_stakeIndex < userStakes[_user].length,Invalid stake index)
(OzoneXStaking.sol#992)
OzoneXStaking.getRewardBreakdown(address,uint256,OzoneXStaking.RewardType)
(OzoneXStaking.sol#1006-1026) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(_stakeIndex < userStakes[_user].length,Invalid stake index)
(OzoneXStaking.sol#1014)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
2 different versions of Solidity are used:
        - Version constraint ^0.8.20 is used by:
                -^0.8.20 (OzoneXStaking.sol#14)
                -^0.8.20 (OzoneXStaking.sol#46)
                -^0.8.20 (OzoneXStaking.sol#148)
                -^0.8.20 (OzoneXStaking.sol#345)
                -^0.8.20 (OzoneXStaking.sol#434)
        - Version constraint >=0.4.16 is used by:
                ->=0.4.16 (OzoneXStaking.sol#262)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
ReentrancyGuard._reentrancyGuardEntered() (OzoneXStaking.sol#425-427) is never used and should be
removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter OzoneXStaking.emergencyWithdrawUSDT(uint256)._amount (OzoneXStaking.sol#1098) is not
in mixedCase
Parameter OzoneXStaking.emergencyWithdrawOZONE(uint256)._amount (OzoneXStaking.sol#1105) is not
in mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Slither:OzoneXStaking.sol analyzed (7 contracts with 93 detectors), 76 result(s) found
```

Solidity Static Analysis

OzoneXStaking.sol

Check-effects-interaction:
Potential violation of the Checks-Effects-Interaction pattern in OzoneXStaking.withdrawOzoneReserves(uint256): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
Pos: 960:4:

Block timestamp:
Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
Pos: 838:34:

Gas costs:
Gas requirement of function OzoneXStaking.deactivatePool is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 663:4:

Gas costs:
Gas requirement of function OzoneXStaking.stake is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 676:4:

Gas costs:
Gas requirement of function OzoneXStaking.fundOzoneReserves is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 941:4:

Gas costs:
Gas requirement of function OzoneXStaking.emergencyWithdrawOZONE is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1105:4:

Similar variable names:
OzoneXStaking.createPool(string,uint256,uint256,uint256,uint256,uint256,bool,bool) : Variables

have very similar names "pools" and "poolId". Note: Modifiers are currently not considered by this static analysis.
Pos: 625:25:

Guard conditions:
Use "assert(x)" if you never want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g., invalid input or a failing external component.
Pos: 686:8:

Data truncated:
Division of integer values yields an integer value again. That means e.g., 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for the division of (only) literal values since those yield rational constants.
Pos: 772:26:

Solhint Linter

OzoneXStaking.sol

```
Compiler version ^0.8.20 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:13
Compiler version ^0.8.20 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:147
Compiler version >=0.4.16 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:261
Compiler version ^0.8.20 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:433
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:567
Avoid making time-based decisions in your business logic
Pos: 16:788
The error message for require is too long
Pos: 13:818
Avoid making time-based decisions in your business logic
Pos: 35:832
Avoid making time-based decisions in your business logic
Pos: 35:837
Possible reentrancy vulnerabilities. Avoid state changes after
transfer.
Pos: 13:847
Possible reentrancy vulnerabilities. Avoid state changes after
transfer.
Pos: 13:848
Possible reentrancy vulnerabilities. Avoid state changes after
transfer.
Pos: 9:853
```

Software analysis result:

This software reported many false positive results, and some are informational issues. So, those issues can be safely ignored.

Email: audit@EtherAuthority.io