SMART CONTRACTS REVIEW

zokyo

March 14th 2024 | v. 1.0

Security Audit Score

PASS

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

ZOKYO AUDIT SCORING BLASTOFF

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

Thus, 100 - 4 = 96

TECHNICAL SUMMARY

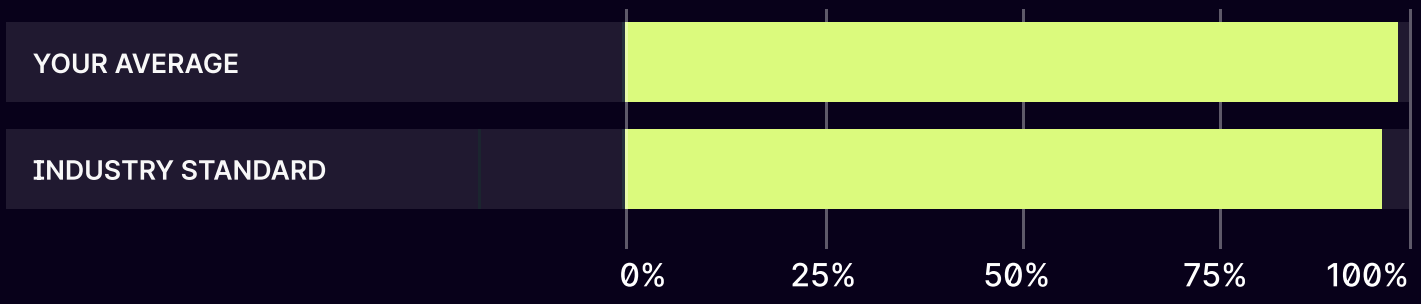
This document outlines the overall security of the BlastOff smart contract/s evaluated by the Zokyo Security team.

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

Contract Status

There was 1 critical issue found during the review. (See Complete Analysis)

Testable Code



98% of the code is testable, which is above the industry standard of 95%.

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 BlastOff 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 BlastOff repository:
Repo: https://github.com/BlastOffOrg/future-yield-contracts

Last commit: d9164fc33ad4de4c6c24792f0354a382069e67d3

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

- ./YieldToken.sol
- ./oracle/ETHPriceOracle.sol
- ./ido/USDIDOPool.sol
- ./ido/ETHIDOPool.sol
- ./ido/IDOPoolAbstract.sol
- ./RoleControl.sol
- ./lib/TokenTransfer.sol
- ./NonLockStakingPool.sol
- ./LockedStakingPools.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 BlastOff 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. Part of this work includes writing a test suite using the Hardhat testing framework. 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.	03	Testing contract/s logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	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 BlastOff team and the BlastOff 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	Unauthorized Withdrawal of Staked ETH via repayWithStakeToken Function	Critical	Resolved
2	Possibility of griefing users via front-running	High	Resolved
3	Incorrect token transfer source in the stake function of NonLockStakingPools contract	High	Resolved
4	Premature Finalization Vulnerability in IDOPoolAbstract Contract	Medium	Resolved
5	Misplaced stake limit validation in stake function of LockedStakingPools contract	Medium	Resolved
6	Incorrect Logic to Prevent Excessive Staking	Medium	Resolved
7	Use of single step ownership transfer	Low	Resolved
8	Use of floating pragma	Low	Acknowledged
9	Misuse of Administrative Privileges	Low	Acknowledged
10	Missing Event Emissions for Critical Actions	Low	Resolved
11	Upgradeable contracts missing method to disable initializers for implementation contracts	Low	Resolved
12	Mutable RoleControl Variable in YieldToken Contract	Informational	Acknowledged
13	Zero Address Check	Informational	Resolved
14	Missing Event Emission	Informational	Resolved
15	Lack of Input Validation for setTokenPriceInUSD Function	Informational	Resolved
16	Redundant Assignment of idoSize in finalize Function	Informational	Resolved

CRITICAL-1 | RESOLVED

Unauthorized Withdrawal of Staked ETH via repayWithStakeToken Function

Affected Function: repayWithStakeToken
Description:
A critical vulnerability allowing attackers to withdraw ETH staked by other users without proper authorization. The function fails to verify the ownership of the stake ID (stakeId) before processing the repayment and unstaking process, enabling anyone who knows the stakeId and the required fee calculation to withdraw the full staked amount by paying only a fraction of its value.

Impact:
Allows unauthorized withdrawal of staked ETH, leading to potential loss of funds for users.

Steps to Reproduce:

1. Identify a target stake ID (stakeId) with a significant amount of staked ETH.
2. Calculate the required fee to repay the stake based on the yield amount and the accumulated yield per staked token. This can be done using the formula provided in the bug description.
3. Call the repayWithStakeToken function with the target poolId, stakeId, and the calculated fee as the transaction value.
4. Observe that the full amount of ETH staked under the target stakeId is transferred to the attacker's address, while only the calculated fee is deducted.

Expected Behavior:
The repayWithStakeToken function should verify that the caller (msg.sender) is the owner of the stake ID (stakeId) before allowing the repayment and unstaking process to proceed. This ensures that only the rightful owner can withdraw the staked ETH.

Actual Behavior:
The function does not perform any ownership verification, allowing any user to withdraw ETH staked by others by exploiting the vulnerability.

Recommendation:
To mitigate this vulnerability, it is recommended to add an ownership check at the beginning of the repayWithStakeToken function. The following line of code should be inserted to ensure that the caller is the owner of the stake:

if (staking.user != msg.sender) revert NotStaker(staking.user);

This check will prevent unauthorized users from exploiting the function to withdraw staked ETH that does not belong to them.

HIGH-1 | RESOLVED

Possibility of griefing users via front-running

The participate function in the LockedStakingPools contract allows users to contribute to a pool with a specified token and amount. This function updates the user's position and the total funded amount for the token. The vulnerability arises from the interaction between the *participate* function and the *_depositToken* internal function. Users must approve the contract to transfer tokens on their behalf before participation.
An attacker can exploit this by front-running a legitimate participation transaction with a call to the participate function using the same token but specifying only 1 wei as the amount. Since the participate function updates the *position.amount* an attacker can cause the legitimate transaction to fail the check if *(position.amount != 0 && token != position.token)* if they manage to execute their front-run transaction first.

Recommendation:
Allow to participate using only the *msg.sender* address as a receiptient.

Incorrect token transfer source in the stake function of NonLockStakingPools contract

The stake function in the NonLockStakingPools contract is designed to allow users to stake tokens into a specific pool, identified by *poolId*. Due to an incorrect source address used in the token transfer operation within the *_depositToken* internal function, called by the stake function. The current implementation attempts to transfer tokens from *address(this)* to itself, rather than transferring tokens from the user *(msg.sender)* who is initiating the stake.

```
poolInfo[poolId].totalStaked += amount;

TokenTransfer._depositToken(pool.stakeToken, address(this), amount);
```

This misimplementation prevents the staking functionality from operating as intended, as the contract does not actually receive tokens from the users.

Recommendation:
The *_depositToken* function's logic should be corrected to transfer tokens from the user *(msg.sender)* to the contract *(address(this))*, rather than attempting to transfer tokens from the contract to itself.

Premature Finalization Vulnerability in IDOPoolAbstract Contract

The IDOPoolAbstract contract suffers from a significant flaw where it allows the contract owner to finalize an Initial DEX Offering (IDO) prematurely without any checks to ensure that predefined conditions, such as reaching an end date or achieving a minimum funding goal, have been met. This lack of validation opens the door to potential manipulation, where the contract owner could finalize the IDO at an opportune moment for personal gain, or prematurely end the IDO, denying participants the full duration to engage as intended.

Recommendation:

Introduce additional conditions within the finalize function to ensure it can only be executed under appropriate circumstances, such as after the IDO has officially ended or once a minimum funding goal has been met. This can be achieved by implementing time-based conditions and/or participation thresholds.

```
// Define IDO start and end times, and minimum funding goal
uint256 public idoStartTime;
uint256 public idoEndTime;
uint256 public minimumFundingGoal;

// Modifier to check that the current time is after the IDO end time
modifier onlyAfterIDOEnd() {
    require(block.timestamp > idoEndTime, "IDO has not ended yet");
    _;
}

// Modifier to check that the minimum funding goal has been met
modifier fundingGoalMet() {
    require(fundedUSDValue >= minimumFundingGoal, "Funding goal not met");
    _;
}

// Update the finalize function to include the new conditions
function finalize() external onlyOwner notFinalized onlyAfterIDOEnd fundingGoalMet {
    // Finalization logic
}
```

MEDIUM-2 | RESOLVED

Misplaced stake limit validation in stake function of LockedStakingPools contract

The following check inside the *stake* function from LockedStakingPools contract attempts to restrict users from having more than 100 active stakes in any given pool.

```
if (userStakeIds[poolId][msg.sender].length == 0) {
  if (userStakeIds[poolId][msg.sender].length > 100) revert TooManyStake();
  noUsersStaked[poolId] += 1;
}
```

However, the check for *userStakeIds[poolId][msg.sender].length > 100* is incorrectly nested within another condition that checks if the user has no existing stakes *(length == 0)*. This contradiction means that the check for exceeding the stake limit is only performed when the user has not yet staked anything (length == 0), which makes it impossible to trigger the *TooManyStake* revert under any circumstance.

Recommendation:

Before adding a new stake, check if the user already has 100 stakes.

MEDIUM-3 | RESOLVED

Incorrect Logic to Prevent Excessive Staking

Description: in LockedStakingPools.sol the logic intended to prevent a user from staking more than 100 times within a pool is flawed due to its placement inside a conditional block that only executes when the user has no previous stakes.

Recommendation:

Move the check for the stake count limit `(userStakeIds[poolId][msg.sender].length > 100)` outside of the conditional block that checks if the user has not staked before `(userStakeIds[poolId][msg.sender].length == 0)`, ensuring it is always evaluated.

LOW-1 | RESOLVED

Use of single step ownership transfer

The IDOPoolAbstract contract in the protocol use the OwnableUpgradeable contract which allows changing the owner address. However, this contract does not implement a 2-step-process for transferring ownership. If the admin's address is set incorrectly, this could potentially result in critical functionalities becoming locked.

Recommendation:
Consider implementing a two-step pattern. Utilize OpenZeppelin's Ownable2StepUpgradeable contract.

LOW-2 | ACKNOWLEDGED

Use of floating pragma

All the contracts in the codebase are using the *pragma solidity ^0.8.20*. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Recommendation:
Consider locking the pragma version in the smart contracts.

LOW-3 | ACKNOWLEDGED

Misuse of Administrative Privileges

Description:
The contract assigns significant control to the pool administrator `(POOL_ADMIN_ROLE)`, including the ability to add new pools, close existing pools, and set the yield APY for pools. While this role is necessary for managing the contract, it introduces a central point of control and trust. An attacker, or a malicious actor with access to the pool admin role, could exploit this control in several ways:

1. **Creation of Duplicate or Similar Pools:** The admin could create multiple pools with identical or very similar terms to existing ones. This could confuse users, leading them to stake in less favorable or intended pools, potentially diluting the staking and yield generation in more legitimate pools.
2. **Prematurely Closing Pools:** An admin could close pools at inopportune times, disrupting users' ability to earn yields and potentially benefiting from the timing of such closures.
3. **Manipulation of Yield APY:** By adjusting the yield APY, an admin could make a pool appear more attractive than it actually is or reduce yields after users have staked a significant amount of tokens.

Recommendation:
Decentralised Governance: Implement a decentralized governance mechanism for critical administrative actions.

LOW-4 | RESOLVED

Missing Event Emissions for Critical Actions

Description:
YieldToken.sol lacks event emissions for key actions such as whitelisting/unwhitelisting addresses and enabling/disabling the whitelist. Events are crucial for off-chain monitoring and transparency of contract operations, especially for actions that impact token transferability and minting operations.

Recommendation:
Define and emit events for all administrative actions, including address whitelisting, unwhitelisting, and changes to the whitelistEnabled flag. This aids in tracking changes and improves the contract's transparency.

LOW-5 | RESOLVED

Upgradeable contracts missing method to disable initializers for implementation contracts

Description:
USDIDOPool.sol ETHIDOPool.sol IDOPoolAbstract.sol RoleControl.sol NonLockStakingPool.sol LockedStakingPools.sol are using initialize() method which uses `initializer` modifier. It is recommended in OpenZeppelin's documentation to not leave implementation contract uninitialised as attacker can take advantage of the same and that may affect the proxy contract.transferability and minting operations.

Recommendation:
Use this suggested method in OpenZeppelin's documentation to mitigate this issue.

INFORMATIONAL-1 | ACKNOWLEDGED

Mutable RoleControl Variable in YieldToken Contract

Description:
The roleControl variable in YieldToken should be immutable for enhanced security but is currently not.

Recommendation:
Declare roleControl as immutable

INFORMATIONAL-2 | RESOLVED

Zero Address Check

Description:
Implement input validation in both setTreasury and setIDOToken functions to reject the zero address. This can be achieved using the require statement to ensure that the provided address is not the zero address,

INFORMATIONAL-3 | RESOLVED

Missing Event Emission

Description:
For each function, define an event that logs the relevant parameters changed by the function call.
setTreasury
setPoolYield
setIDOToken
setTokenPriceInUSD
setEnableWhitelist

INFORMATIONAL-4 | RESOLVED

Lack of Input Validation for setTokenPriceInUSD Function

Description:
The setTokenPriceInUSD function in the IDOPoolAbstract contract lacks input validation, allowing for the setting of an IDO price to any value without restrictions. This poses a risk as it enables the contract owner to arbitrarily change the IDO price to potentially unfair or nonsensical values.

Recommendation:
Implement input validation within the setTokenPriceInUSD function to ensure that the IDO price is within acceptable bounds. Use require statements to enforce these conditions and provide meaningful error messages for invalid inputs.

Redundant Assignment of idoSize in finalize Function

Description:
In the finalize function of the IDOPoolAbstract contract, the idoSize variable is assigned the balance of idoToken held by the contract twice in succession without any operation in between that would change the balance. This redundancy suggests a potential oversight in the contract's logic, where either the double assignment is unnecessary, or it indicates a missing piece of logic that should have been implemented between these two assignments.

Recommendation:
Review the intended logic for the finalize function to determine if the double assignment of idoSize is necessary. If it is redundant, remove the first assignment to clean up the code and prevent potential confusion. If there was intended to be additional logic between these assignments that was omitted, implement the missing logic to ensure the function operates as expected.

	./YieldToken.sol ./oracle/ETHPriceOracle.sol ./ido/USDIDOPool.sol ./ido/ETHIDOPool.sol ./ido/IDOPoolAbstract.sol
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

	./RoleControl.sol ./lib/TokenTransfer.sol ./NonLockStakingPool.sol ./LockedStakingPools.sol
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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting BlastOff in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the BlastOff contract/s requirements for details about issuance amounts and how the system handles these.

```
############ IDOPools ############
    === ETHIDOPool ===
    ✓ participate() -- balances and account positions updated as expected (86ms)
    ✓ participate() -- twice participate (97ms)
    ✓ participate() -- revert; participate second time with wrong choice of token (77ms)
    ✓ participate() -- revert; participate with invalid token (39ms)
    ✓ participate() -- revert; after being finalized (231ms)
    ✓ claim() -- Confirm user receives the idoToken (281ms)
    ✓ withdrawSpareIDO() -- Confirm admin receives the idoToken (258ms)
    ✓ withdrawSpareIDO() -- less fundedUSDValue (248ms)
    ✓ claim() -- revert; cannot claim before being finalized (55ms)
    ✓ claim() -- revert; cannot claim without staking (196ms)
    ✓ setIDOToken() -- set the idoToken
    ✓ setIDOToken() -- revert; not admin
    ✓ setTokenPriceInUSD() -- revert; not admin
    ✓ finalize() -- revert; not admin
    ✓ withdrawSpareIDO() -- revert; not admin (204ms)
    ✓ withdrawSpareIDO() -- revert; not admin
    ✓ init() --- revert; not allowed to init second time
    === USDIDOPool ===
    ✓ participate() -- balances and account positions updated as expected (61ms)
    ✓ participate() -- twice participate (80ms)
    ✓ participate() -- revert; participate second time with wrong choice of token (55ms)
    ✓ participate() -- revert; participate with invalid token
    ✓ participate() -- revert; after being finalized (78ms)
    ✓ finalize() -- revert; after being already finalized (125ms)
    ✓ init() --- revert; not allowed to init second time
```

```
########### LockedStakingPools ###########
```

stake() scenarios:

✓ stake() --- expect yield increase by stake amount (78ms)
✓ stake() --- USDB as yield (147ms)
✓ stake() --- staking second consecutive time (123ms)
✓ stake() --- revert due to invalid Arguments
✓ stake() --- revert due to PoolClosed
✓ stake() --- revert; pool is not enabled (38ms)
✓ stake() --- revert due to mismatch between msg.value and amount (55ms)

extendsPosition() scenarios:

✓ extendPosition() --- expect unlockTime increases and YieldBalance (112ms)
✓ extendsPosition() --- revert; PoolNotExisted
✓ extendPosition() --- revert; NoStaking (38ms)
✓ extendPosition() --- revert; NotStaker (81ms)
✓ extendPosition() --- revert; just more than ten year extension (79ms)
✓ extendPosition() --- edge case: exactly ten year after extension (97ms)

unstake() scenarios:

✓ unstake() --- successful unstake; staker takes expected amount of assets back (104ms)
✓ unstake() --- revert; PoolNotExisted
✓ unstake() --- revert; NoStaking
✓ unstake() --- revert; NotStaker (84ms)
✓ unstake() --- revert; Locked (77ms)

repayWithStakeToken() scenarios:

✓ repayWithStakeToken() --- Staker should receive back the staking token (115ms)
✓ repayWithStakeToken() --- Staker repays only requireAmount, leftover returned (105ms)
✓ repayWithStakeToken() --- revert; for paying insuffcient msg.value (104ms)
✓ repayWithStakeToken() --- revert; PoolNotExisted
✓ repayWithStakeToken() --- revert; NoStaking

repayWithYieldToken() scenarios:

✓ repayWithYieldToken() --- Staker should receive back the staking token (128ms)
✓ repayWithStakeToken() --- revert; PoolNotExisted
✓ repayWithStakeToken() --- revert; NoStaking

other scenarios:

✓ setPoolYield() -- value updated (65ms)
✓ setPoolYield() -- revert; (61ms)
✓ setPoolYield() -- revert; not called by admin (56ms)
✓ setPoolYield() -- revert; Invalid Arguments (58ms)
✓ setPoolYield() -- revert; PoolNotExisted
✓ closePool() -- revert; not called by admin
✓ closePool() -- confirm pool deleted
✓ addLockedPools() -- revert; Invalid Arguments

✓ addLockedPools() -- revert; Invalid stake token
✓ addLockedPools() -- revert; not called by admin
✓ addSupportYieldTokens() -- revert; not called by admin
✓ setTreasury() -- revert; not called by admin
✓ getGeneratedYield() -- (230ms)
✓ stakeTimes() -- stake 2x (178ms)
✓ claimNativeYield() --- admin successfully claim yield
✓ claimNativeYield() --- revert; unauthorized caller
✓ claimUSDBYield() --- admin successfully claim yield
✓ claimUSDBYield() --- revert; unauthorized caller
✓ setBlast() --- revert; unauthorized caller
✓ setUSDBRebasing() --- revert; unauthorized caller
✓ init() --- revert; not allowed to init second time
✓ RoleControl::init() --- revert; not allowed to init second time
✓ addLockedPools() --- revert due to invalid Arguments
✓ addSupportYieldTokens() --- revert; invalid arguments

########## NonLockStakingPools ##########

stake() scenarios:

✓ stake() --- expect native coin to increase by stake amount for subject being tested
✓ stake() --- epxect stakes to add up (50ms)
✓ stake() --- revert due to PoolNotExisted

unstake() scenarios:

✓ unstake() --- expected staking balances (61ms)
✓ unstake() --- revert due to PoolNotExisted

claimPendingReward() scenarios:

✓ claimPendingReward() --- expected rewards (70ms)
✓ claimPendingReward() --- revert due to PoolNotExisted
✓ claimPendingReward() --- revert due to NoStaking

other scenarios:

✓ setPoolYield() -- value updated (59ms)
✓ setPoolYield() -- revert; not called by admin (56ms)
✓ setPoolYield() -- revert; Invalid Arguments (54ms)
✓ setPoolYield() -- revert; PoolNotExisted
✓ closePool() -- revert; not called by admin
✓ closePool() -- confirm pool deleted
✓ addPool() -- revert; Invalid Arguments
✓ addPool() -- revert; Invalid stake token
✓ addPool() -- revert; not called by admin
✓ addSupportYieldTokens() -- revert; not called by admin
✓ setTreasury() -- revert; not called by admin
✓ init() --- revert; not allowed to init second time

```
############ YieldToken ############
    scenarios:
    ✓ unwhitelistAddress() --- expect address deleted from whitelist
    ✓ unwhitelistAddress() --- revert; caller not admin
    ✓ whitelistAddress() --- revert; caller not admin
    ✓ setEnableWhitelist() --- expect whitelistEnabled be true
    ✓ setEnableWhitelist() --- revert; caller not admin
    ✓ mint() --- revert; caller not minter role

    100 passing (32s)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	%UNCOVERED LINES
LockedStakingPools.sol	99.04	87.76	100	98.54	413, 461
NonLockStakingPool.sol	97.3	89.47	91.67	98.28	47
RoleControl.sol	100	50	100	100	
YieldToken.sol	100	78.57	100	92.31	20
ETHIDOPool.sol	100	100	100	100	
IDOPoolAbstract.sol	96.77	88.89	100	98.04	110
USDIDOPool.sol	100	100	100	100	
TokenTransfer.sol	100	100	83.33	100	
ETHPriceOracle.sol	75	50	100	100	
All Files	**98.01**	**86.54**	**98.48**	**98.23**	

We are grateful for the opportunity to work with the BlastOff 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 BlastOff team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo