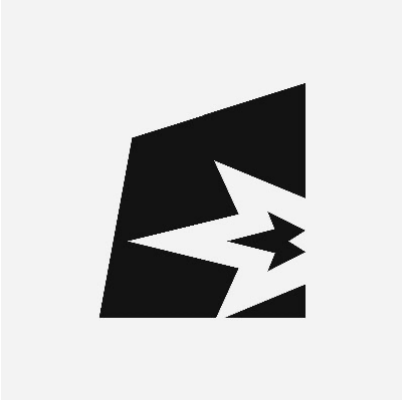
verichains

SECURITY AUDIT OF

IDO POOLS



Private Report

Jul 16, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.
Blast	Blast Blockchain is a Fast, Secure, Scalable, Low-cost, Layer two (L2) and EVM compatible Blockchain built on the Ethereum network. It is built on the Optimism rollup, inheriting the powerful OP Stack. Blast stands out from other L2 network due to its native yield feature.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jul 16, 2024. We would like to thank the BlastOff for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the IDO Pools. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team found some vulnerabilities in the given version of IDO Pools. BlastOff team has acknowledged all issues.

CONFIDENTIALITY NOTICE

This report may contain privileged and confidential information, or information of a proprietary nature, and information on vulnerabilities, potential impacts, attack vectors of vulnerabilities which were discovered in the process of the audit.

The information in this report is intended only for the person to whom it is addressed and/or otherwise authorized personnel of BlastOff. If you are not the intended recipient, you are hereby notified that you have received this document in error, and that any review, dissemination, printing, or copying of this message is strictly prohibited. If you have received this communication in error, please delete it immediately.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About IDO Pools

IDO Pools are a revolutionary new type of initial decentralized offering (IDO) that combines the traditional concept of IDOs with the innovative concept of native yield. In a nutshell, IDO Pools allow users to participate in IDO without the risk of losing their investment. This is achieved by allocating a portion of the yield generated from staking ETH or stablecoins on the BlastOff platform to support.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of IDO Pools. It was conducted on the commit `10a925cdcd69ae433074bf17580fdbf8df442630` from git repository *https://github.com/BlastOffOrg/Ido-Pools-Foundry*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
`b88e74d2971ec71740b572b0ecae121a87daf7118175e2738e4bd650162866f4`	`src/ETHIDOPool.sol`
`9855d389aea0d30c88f427c7bece6dffb8d2b27b18f19260a14e2d59680ff8b0`	`src/IDOPoolAbstract.sol`
`efa78e37cd50f124195547a7689f80a44b12e1d8f5c90e2742ba13a804488f35`	`src/USDIDOPool.sol`

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert

- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

BlastOff acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. BlastOff understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, BlastOff agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the BlastOff will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the BlastOff, the final report will be considered fully accepted by the BlastOff without the signature.

2. AUDIT RESULT

2.1. Overview

The IDO Pools was written in `Solidity` language, with the required version to be `0.8.20`.

This smart contract facilitates an Initial DEX Offering (IDO) where participants can buy tokens using either ETH or USD. It inherits from `Ownable2StepUpgradeable` and implements the `IIDOPool` interface, managing the IDO lifecycle including participation, finalization, and token claiming. The contract tracks various parameters such as IDO token details, funding goals, and timeframes. It uses modifiers to ensure actions occur at appropriate stages, e.g., after the start or upon finalization. Participants' positions are recorded, enabling calculation of their token allocations and excess funds. Upon finalization, funds are managed, and participants can claim their tokens or refunds accordingly. The contract also supports withdrawing unallocated IDO tokens if the funding goal is unmet.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of IDO Pools. BlastOff team acknowledged these issues according to Verichains's draft report. The following table shows the vulnerabilities found during the audit:

Title	Severity	Status
Wrong implementation in excessive USD calculation	CRITICAL	ACKNOWLEDGED
Unexpected revert when finalizing IDO	CRITICAL	FIXED
Incorrect implementation in `withdrawSpareIDO`	HIGH	ACKNOWLEDGED
`claimableTime` is not initialized	HIGH	FIXED
Incorrect rounding up in `_getPostionValue`	MEDIUM	ACKNOWLEDGED
Incorrect requirement in `finalize` function	MEDIUM	FIXED
Unused variable and modifier in `IDOPoolAbstract`	LOW	FIXED
Not use `__Ownable2Step_init` instead of `__Ownable_init` in `IDOPoolAbstract`	LOW	FIXED
Arbitrary input decimal of ido token	LOW	FIXED

Table 2. Vulnerability List

2.2.1. CRITICAL - Wrong implementation in excessive USD calculation

Positions:

- IDOPoolAbstract.sol#L155

Description:

The `excessiveInUSD` calculation is incorrect. In the contract, The `excessiveInUSD` is calculated as the difference between the `posInUSD` and the `exceedAlloc` multiplied by the `idoExp` divided by the `idoPrice`. Unfortunately, this calculation is incorrect.

```
function _getPostionValue(
    Position memory pos
) internal view returns (uint256 allocated, uint256 excessive) {
    //...
    if (((idoSize * idoPrice) / idoExp) >= fundedUSDValue) {
        return (buyAlloc, 0);
    } else {
        uint256 excessiveInUSD = posInUSD - ((exceedAlloc * idoExp) / idoPrice); //
Incorrect calculation
        return (
            exceedAlloc,
            (excessiveInUSD * snapshotPriceDecimals) / snapshotTokenPrice
        );
    }
}
```

RECOMMENDATION

The `excessiveInUSD` should be calculated as the difference between the `posInUSD` and the `exceedAlloc` multiplied by the `idoPrice` divided by the `idoExp`.

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.

2.2.2. CRITICAL - Unexpected revert when finalizing IDO

Positions:

- IDOPoolAbstract.sol#L125

Description:

The `idoSize` state is `zero` before the `finalize()` function is called. Therefore, the function reverts with the `FudingGoalNotReached` error because the `idoSize` is always less than the `minimumFundingGoal`.

```
function finalize() external onlyOwner notFinalized {
    if (block.timestamp < idoEndTime) revert IDONotEnded();
    else if (idoSize < minimumFundingGoal) revert FudingGoalNotReached();
    //...
  }
```

RECOMMENDATION

The `idoSize` should be updated before minimum funding goal check.

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.
- ***Jul 16, 2024***: The issue has been fixed by the BlastOff team.

2.2.3. HIGH - Incorrect implementation in `withdrawSpareIDO`

Position:

- `IDOPoolAbstract.sol`#withdrawSpareIDO()

Description:

The `withdrawSpareIDO` function is implemented to withdraw the remaining IDO tokens after the IDO is finalized. However, the ido balance is dynamic and subject to alter based on user claims. As a result, the function may not work as expected because ido balance is not enough to withdraw. Additionally, the function cannot be called if finalized is not invoked.

```
function withdrawSpareIDO() external finalized onlyOwner {
        uint256 totalIDOGoal = (idoSize * idoPrice) / (10 ** idoDecimals);
        if (totalIDOGoal <= fundedUSDValue) revert();

        uint256 totalBought = fundedUSDValue / idoPrice * (10 ** idoDecimals);
        uint256 idoBal = IERC20Mintable(idoToken).balanceOf(address(this));
        uint256 spare = idoBal - totalBought;
        TokenTransfer._transferToken(idoToken, msg.sender, spare);
    }
```

RECOMMENDATION

Carefully review and change the implementation of the `withdrawSpareIDO` function.

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.

2.2.4. HIGH - `claimableTime` is not initialized

Position:

- `IDOPoolAbstract.sol`#L17

Description:

The `claimableTime` state is not initialized in the initialization function. Therefore, the `claimableTime` is always zero.

```
uint256 public claimableTime;

    modifier claimable() {
        if (!isFinalized) revert NotFinalized();
        if (block.timestamp < claimableTime) revert NotClaimable();
        _;
    }
```

RECOMMENDATION

The `claimableTime` should be initialized in the initialization function.

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.
- ***Jul 16, 2024***: The issue has been fixed by the BlastOff team.

2.2.5. MEDIUM - Incorrect rounding up in `_getPostionValue`

Position:

- `IDOPoolAbstract.sol`#_getPostionValue()

Description:

In the `_getPostionValue` function, the `excessiveInUSD` is rounded up when calculated. As a result, the last user's token getting stuck.

```
function _getPostionValue(
    Position memory pos
  ) internal view returns (uint256 allocated, uint256 excessive) {
    //...
    if (((idoSize * idoPrice) / idoExp) >= fundedUSDValue) {
      return (buyAlloc, 0);
    } else {
      uint256 excessiveInUSD = posInUSD - ((exceedAlloc * idoExp) / idoPrice); // Incorrect rounding up
      return (
        exceedAlloc,
        (excessiveInUSD * snapshotPriceDecimals) / snapshotTokenPrice
      );
    }
  }
```

RECOMMENDATION

Always round down the amount transferred to the user.

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.

2.2.6. MEDIUM - Incorrect requirement in `finalize` function

Position:

- `IDOPoolAbstract.sol`#finalize()

Description:

The `idoSize` should be hard-coded when initializing the contract. However, the contract get the `idoSize` from contract balance when finalization. This is incorrect because the `idoSize` is dynamic. The minimum funding goal should be checked by the `fundedUSDValue` or sum of amount `buytoken` and `fytoken` instead of `idoSize`.

```
function finalize() external onlyOwner notFinalized {
    if (block.timestamp < idoEndTime) revert IDONotEnded();
    else if (idoSize < minimumFundingGoal) revert FudingGoalNotReached(); // Incorrect
requirement
    //...
  }
```

RECOMMENDATION

Carefully review and change the implementation of the `finalize` function.

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.
- ***Jul 16, 2024***: The issue has been fixed by the BlastOff team.

2.2.7. LOW - Unused variable and modifier in `IDOPoolAbstract`

Position:

- `IDOPoolAbstract.sol`#L18
- `IDOPoolAbstract.sol`#L45

Description:

The `isClaimable` state and `notStart` modifier are not used in the contract. Therefore, these should be removed from the contract.

```
bool public isClaimable;
    modifier notStart() {
        if (block.timestamp >= idoStartTime) revert AlreadyStarted();
        _;
    }
```

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.
- ***Jul 16, 2024***: The issue has been fixed by the BlastOff team.

2.2.8. LOW - Not use `__Ownable2Step_init` instead of `__Ownable_init` in `IDOPoolAbstract`

Position:

- `IDOPoolAbstract.sol`#L77

Description:

When the contract use `Ownable2StepUpgradeable` abstract, the `__Ownable_init` should be replaced with `__Ownable2Step_init`.

```
function __IDOPoolAbstract_init(
    //...
  ) internal onlyInitializing {
    //...
    __Ownable_init(); // Wrong initialization
  }
```

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.
- ***Jul 16, 2024***: The issue has been fixed by the BlastOff team.

2.2.9. LOW - Arbitrary input decimal of ido token

Position:

- IDOPoolAbstract.sol#L94
- IDOPoolAbstract.sol#L108

Description:

The owner can input wrong decimal of `idoToken`. Therefore, the decimal should be got from contract by `decimals()` function.

```
function setIDOToken(
        address _token,
        uint256 _idoDecimals
    ) external onlyOwner {
        //...
        idoDecimals = _idoDecimals; // Incorrect
    }

    function __IDOPoolAbstract_init_unchained(
        //...
    ) internal onlyInitializing {
        //...
        idoDecimals = idoDecimals_; // Incorrect
        //...
    }
```

UPDATES

- ***Jul 09, 2024***: The issue has been acknowledged by the BlastOff team.
- ***Jul 16, 2024***: The issue has been fixed by the BlastOff team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jul 09, 2024*	Private Report	Verichains Lab
1.1	*Jul 16, 2024*	Private Report	Verichains Lab

Table 3. Report versions history