Holofair Token Launchpad

Security Review

Disclaimer

Security Review

Holofair Token Launchpad

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

Holofair Token Launchpad

Table of Contents

Summary

Security Review

Holofair Token Launchpad

Summary

Three Sigma audited Holofair in a 1.2 person week engagement. The audit was conducted from 06/05/2025 to 08/05/2025.

Protocol Description

Holofair is a decentralized launchpad for deploying ERC20 tokens with automated presale mechanisms and liquidity provisioning. It integrates with Uniswap V2 to ensure seamless liquidity management and enables fair token distributions.

Scope

Security Review

Holofair Token Launchpad

Scope

Filepath	nSLOC
src/HoloFair.sol	217
src/HolofairToken.sol	196
Total	**413**

Assumptions

OpenZeppelin library is considered secure.

Methodology

Security Review

Holofair Token Launchpad

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Holofair Token Launchpad

Project Dashboard

Application Summary

Name	Holofair
Repository	https://github.com/roy105703007/Holofair
Commit	2a82fb9
Language	Solidity
Platform	Abstract

Engagement Summary

Timeline	06/05/2025 to 08/05/2025
№ of Auditors	2
Review Time	1.2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	1	1	0
Medium	0	0	0
Low	1	1	0
None	2	2	0

Category Breakdown

Suggestion	2
Documentation	0
Bug	2
Optimization	0
Good Code Practices	0

Risk Section

Security Review

Holofair Token Launchpad

Risk Section

No risks Identified.

Findings

Security Review

Holofair Token Launchpad

Findings

3S-Holofair-H01

Missing **whitelistTotalDeposited** update in withdrawal flow enables DoS of presale whitelist allocation

Id	3S-Holofair-H01
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #9c44dd6.

Description

The **HolofairToken** contract implements a two-phase presale system with separate allocation tracking for whitelist and public participants. The contract maintains several accounting variables to track deposits:

- **deposits[address]**: Tracks total deposits per user

- **whitelistDeposits[address]**: Tracks whitelist-specific deposits per user

- **whitelistTotalDeposited**: Global counter for all whitelist deposits

- **publicTotalDeposited**: Global counter for all public deposits

When users deposit ETH via the **depositWhitelist()** function, the contract properly updates both user-specific and global accounting variables:

```
whitelistTotalDeposited += depositAmount;
whitelistDeposits[msg.sender] += depositAmount;
```

However, in the **withdrawDeposit()** function, while the contract correctly updates **whitelistDeposits[msg.sender]** and other accounting variables, it fails to decrement the **whitelistTotalDeposited** global counter:

```
function withdrawDeposit(uint256 amount) external nonReentrant {
```

```
    require(
        presaleStatus == Status.Phase1 || presaleStatus == Status.Phase2,
        InvalidStatusNotActive()
    );
    uint256 userDeposit = deposits[msg.sender];
    require(userDeposit >= amount, NoDeposit());
    if (presaleStatus == Status.Phase1) {
        // Calculate proportion of whitelist deposit to withdraw
        if (whitelistDeposits[msg.sender] > 0) {
            if (whitelistDeposits[msg.sender] >= amount) {
                whitelistDeposits[msg.sender] -= amount;
            } else {
                publicTotalDeposited -= (amount -
                    whitelistDeposits[msg.sender]);
                whitelistDeposits[msg.sender] = 0;
            }
        } else {
            publicTotalDeposited -= amount;
        }
    }
    // Missing: whitelistTotalDeposited is not updated
    deposits[msg.sender] -= amount;
    totalRaised -= amount;
    // ...
}
```

This accounting error enables a malicious whitelisted user to permanently block whitelist allocation by repeatedly depositing and withdrawing ETH. Each deposit-withdraw cycle artificially inflates the **whitelistTotalDeposited** counter without actually consuming real capacity. Eventually, the counter will reach the **whitelistCap** limit, making any further legitimate whitelist deposits impossible due to the check in **depositWhitelist()**:

```
require(
    whitelistTotalDeposited + depositAmount <= whitelistCap,
    ExceedsWhitelistTotalCap()
);
```

Steps to Reproduce:

1. A malicious user obtains a valid merkle proof for the whitelist with some **addressMaxDeposit** value.

2. User calls **HolofairToken::depositWhitelist()** with 1 ETH.

3. **whitelistTotalDeposited** is incremented by 1 ETH.

4. User immediately calls **HolofairToken::withdrawDeposit()** to withdraw the 1 ETH.

5. The user's **whitelistDeposits** mapping is decremented, but **whitelistTotalDeposited** remains unchanged.

6. User repeats steps 2-5 multiple times until **whitelistTotalDeposited** approaches or equals **whitelistCap**.

7. Legitimate whitelist users attempt to deposit but their transactions revert with **ExceedsWhitelistTotalCap()** even though actual deposits are much lower.

Recommendation

Update the **withdrawDeposit()** function to properly decrement the **whitelistTotalDeposited** counter when whitelist deposits are withdrawn:

```
function withdrawDeposit(uint256 amount) external nonReentrant {
    require(
        presaleStatus == Status.Phase1 || presaleStatus == Status.Phase2,
        InvalidStatusNotActive()
    );
    uint256 userDeposit = deposits[msg.sender];
    require(userDeposit >= amount, NoDeposit());
    if (presaleStatus == Status.Phase1) {
        // Calculate proportion of whitelist deposit to withdraw
        if (whitelistDeposits[msg.sender] > 0) {
            if (whitelistDeposits[msg.sender] >= amount) {
                whitelistDeposits[msg.sender] -= amount;
+                whitelistTotalDeposited -= amount;
            } else {
+                whitelistTotalDeposited -= whitelistDeposits[msg.sender];
                publicTotalDeposited -= (amount -
                    whitelistDeposits[msg.sender]);
                whitelistDeposits[msg.sender] = 0;
            }
        } else {
            publicTotalDeposited -= amount;
        }
    }
    deposits[msg.sender] -= amount;
    totalRaised -= amount;
    // ...
```

```
}
```

This fix ensures that **whitelistTotalDeposited** accurately reflects the actual amount deposited by whitelist participants at all times, preventing the DoS attack vector.

3S-Holofair-L01

Missing whitelist deposit check in Phase 2 allows exceeding **addressMaxDeposit**

Id	3S-Holofair-L01
Classification	Low
Impact	Low
Likelihood	Medium
Category	Bug
Status	Addressed in #03c2bfc.

Description

The **HolofairToken** contract is designed to manage a presale with two phases. In Phase 1, whitelist users can deposit up to their custom **addressMaxDeposit** limit using the **HolofairToken::depositWhitelist** function. In Phase 2, all users, including those who were part of the whitelist, can deposit using the **HolofairToken::receive** function. However, the contract fails to enforce the **addressMaxDeposit** limit for whitelist users in Phase 2. This oversight allows whitelist users to exceed their **addressMaxDeposit** limit by depositing additional funds in Phase 2, which contradicts the project documentation stating: "If a whitelist user has already minted more than general per-wallet cap in Phase 1, they will not be able to mint in Phase 2."

Recommendation

To address this issue, the **HolofairToken::receive** function should include a check during Phase 2 to ensure that the total deposits of a whitelist user do not exceed their **addressMaxDeposit**. The following code diff demonstrates the necessary change:

```
function receive() external payable depositAllowed {
    uint256 depositAmount = msg.value;
    if (presaleStatus == Status.Phase1) {
        // Phase 1 logic
```

```
        require(depositAmount <= maxDeposit, ExceedsPublicWalletCap());
        require(publicTotalDeposited + depositAmount <= publicCap,
ExceedsPublicTotalCap());
        publicTotalDeposited += depositAmount;
    } else {
        // Phase 2 logic
+       require(whitelistDeposits[msg.sender] + depositAmount <=
addressMaxDeposit, ExceedsPublicWalletCap());
        require(depositAmount <= maxDeposit, ExceedsPublicWalletCap());
    }
    _deposit(depositAmount);
}
```

This modification ensures that the total deposits of a whitelist user, including those made in Phase 1, do not exceed their **addressMaxDeposit** limit during Phase 2.

3S-Holofair-N01

Redundant **maxDeposit** check in receive function allows unnecessary code duplication

Id	3S-Holofair-N01
Classification	None
Category	Suggestion
Status	Addressed in #fb3e8b6.

Description

The **HolofairToken::receive** function is designed to handle incoming Ether deposits during the presale phases. It checks the presale status and applies different logic for Phase 1 and Phase 2. However, the check for **maxDeposit** is redundantly placed in both the **if** and **else** branches of the conditional statement. This redundancy does not affect the functionality but leads to unnecessary code duplication, which can reduce code readability and maintainability.

```
    receive() external payable depositAllowed {
        uint256 depositAmount = msg.value;
        if (presaleStatus == Status.Phase1) {
            // Phase 1 logic
            require(depositAmount <= maxDeposit, ExceedsPublicWalletCap());
            require(publicTotalDeposited + depositAmount <= publicCap,
ExceedsPublicTotalCap());
            publicTotalDeposited += depositAmount;
        } else {
            // Phase 2 logic
            require(depositAmount <= maxDeposit, ExceedsPublicWalletCap());
        }
        _deposit(depositAmount);
    }
```

Recommendation

To improve code quality and maintainability, the **maxDeposit** check should be moved outside the conditional statement, as it applies to both presale phases. This change will eliminate redundancy and make the code cleaner.

3S-Holofair-N02

Unprecise minimum value check for **whitelistReserveTime** leads to unintended Phase 1 activation

Id	3S-Holofair-N02
Classification	None
Category	Suggestion
Status	Addressed in #878c856.

Description

The **HolofairToken** contract is designed to manage a presale process with distinct phases, including a whitelist phase (Phase 1) and a public phase (Phase 2). The **whitelistReserveTime** parameter is intended to define the duration of Phase 1. However, the contract does not enforce a minimum value for **whitelistReserveTime**, allowing it to be set to zero. When **whitelistReserveTime** is zero, the **whitelistReserveEndTime** is initialized to the current block timestamp during the presale start. Consequently, the condition in the **depositAllowed** modifier, which checks if the current time has surpassed **whitelistReserveEndTime**, fails to transition immediately to Phase 2. This results in Phase 1 being erroneously activated for the first transaction, contrary to the intended behavior.

Recommendation

To ensure that Phase 1 is skipped when **whitelistReserveTime** is zero, modify the condition in the **depositAllowed** modifier to transition to Phase 2 when the current block timestamp is equal to or greater than **whitelistReserveEndTime**.

```
- if (presaleStatus == Status.Phase1 && block.timestamp > whitelistReserveEndTime)
{
+ if (presaleStatus == Status.Phase1 && block.timestamp >=
whitelistReserveEndTime) {
```