Prepared by: 0xshrxyeh

Disclaimer

Shreyash Naik makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the team 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.

Table of contents

Risk Classification

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

Audit Details

Scope

```
./src/
-- ChristmasDinner.sol
```

Protocol Summary

This contract is designed as a modified fund me. It is supposed to sign up participants for a social christmas dinner (or any other dinner), while collecting payments for signing up.

Roles

-`Host`: The person doing the organization of the event. Receiver of the funds by the end of `deadline`. Privilegded Role, which can be handed over to any `Participant` by the current `host` - `Participant`: Attendees of the event which provided some sort of funding. `Participant` can become new `Host`, can continue sending money as Generous Donation, can sign up friends and can become `Funder`. -`Funder`: Former Participants which left their funds in the contract as donation, but can not attend the event. `Funder` can become `Participant` again BEFORE deadline ends.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	2
Info	1
Total	10

Findings

High Risk Findings

[H-01] Ineffective Reentrancy Protection in `nonReentrant` Modifier

Description:
The `nonReentrant` modifier's implementation is flawed as it fails to set the `locked` state variable to `true` at the beginning of the function execution. This creates a window of vulnerability where reentrancy attacks could be possible.

```
modifier nonReentrant() {
    require(!locked, "No re-entrancy");
    _; // Lock is not set to true before execution
    locked = false;
}
```

Impact: - Potential reentrancy attacks could drain funds from the contract - Multiple simultaneous calls to protected functions could succeed - Critical functions like `refund()` are not properly protected

Proof of Concept:

```
function testReentrancyAttack() public peopleEntered {
        // Deploy attacker contract
        AttackerContract attacker = new AttackerContract(
            address(payable(cd)),
            address(wbtc)
        );

        // Fund and approve attacker
        wbtc.mint(address(attacker), 3 ether);

        // Record initial states
        uint256 initialContractBalance =
        wbtc.balanceOf(address(cd));
        uint256 initialAttackerBalance =
        wbtc.balanceOf(address(attacker));

        console.log("Initial contract WBTC balance:",
        initialContractBalance);
        console.log("Initial attacker WBTC balance:",
        initialAttackerBalance);

        // Execute attack
        vm.prank(address(attacker));
        attacker.attack();

        // Verify attack results
        uint256 finalContractBalance =
        wbtc.balanceOf(address(cd));
        uint256 finalAttackerBalance =
        wbtc.balanceOf(address(attacker));

        console.log("Final contract WBTC balance:",
        finalContractBalance);
        console.log("Final attacker WBTC balance:",
        finalAttackerBalance);

        // Assert the attack was successful
        assertGt(
            finalAttackerBalance,
            initialAttackerBalance,
            "Attack didn't increase attacker's balance"
        );
        assertLt(
            finalContractBalance,
            initialContractBalance,
```

```
            "Contract balance wasn't drained"
        );
    }
```

Recommended Mitigation: Implement the reentrancy guard correctly by setting the lock before function execution:

```
modifier nonReentrant() {
    require(!locked, "No re-entrancy");
    locked = true;
    _;
    locked = false;
}
```

[H-02]Reentrancy Vulnerability in `_refundERC20` Function

Description:
The `_refundERC20` function in the `ChristmasDinner` contract transfers ERC20 tokens to the user before updating their balance to zero. This introduces a reentrancy vulnerability because the external call to `safeTransfer` allows the recipient to re-enter the contract before the balance is reset. If exploited, this can drain the entire contract balance.

Impact:
- Attackers can repeatedly call the `refund()` function through a fallback or receive function.
- This could result in the attacker draining all ERC20 tokens from the contract.
- The contract's funds can be entirely compromised, causing a significant financial loss.

Vulnerable Code:

```
function _refundERC20(address _to) internal {
    i_WETH.safeTransfer(_to, balances[_to][address(i_WETH)]);
    i_WBTC.safeTransfer(_to, balances[_to][address(i_WBTC)]);
    i_USDC.safeTransfer(_to, balances[_to][address(i_USDC)]);
    balances[_to][address(i_USDC)] = 0;
    balances[_to][address(i_WBTC)] = 0;
    balances[_to][address(i_WETH)] = 0;
}
```

Recommended Mitigation:

```
function _refundERC20(address _to) internal {
    uint256 wethAmount = balances[_to][address(i_WETH)];
    uint256 wbtcAmount = balances[_to][address(i_WBTC)];
    uint256 usdcAmount = balances[_to][address(i_USDC)];
```

```
    balances[_to][address(i_USDC)] = 0;
    balances[_to][address(i_WBTC)] = 0;
    balances[_to][address(i_WETH)] = 0;

    i_WETH.safeTransfer(_to, wethAmount);
    i_WBTC.safeTransfer(_to, wbtcAmount);
    i_USDC.safeTransfer(_to, usdcAmount);
}
```

[H-03] Reentrancy Vulnerability in `_refundETH` Function

Description:
The `_refundETH` function transfers ETH to the user before updating their balance to zero. This creates a reentrancy vulnerability because the external call `_to.transfer(refundValue)` allows the recipient to re-enter the contract before their balance is set to zero. Additionally, the use of `.transfer` has a fixed gas limit of 2300, which can cause transactions to fail if more gas is required.

Impact:
- Attackers can re-enter the contract and repeatedly call `refund()`, draining the ETH balance.
- This can lead to complete depletion of ETH in the contract.
- The vulnerability is severe and can cause significant financial loss if exploited.

Vulnerable Code:

```
function _refundETH(address payable _to) internal {
    uint256 refundValue = etherBalance[_to];
    _to.transfer(refundValue);
    etherBalance[_to] = 0;
}
```

Recommended Mitigation:

```
function _refundETH(address payable _to) internal {
    uint256 refundValue = etherBalance[_to];
    etherBalance[_to] = 0;  // Update balance before external
call

    (bool success, ) = _to.call{value: refundValue}("");
    require(success, "ETH Transfer Failed");
}
```

[H-04] Unprotected Withdrawal Function Allows Premature Fund Drainage

Description: The `withdraw()` function lacks a deadline check, allowing the host to withdraw funds while the contract is still accepting deposits.

Impact: - Host can drain contract while deposits are still allowed - Users could lose funds by depositing after withdrawal - Breaks the trust assumption of the contract

Proof of Concept:

```
function withdraw() external onlyHost {
    address _host = getHost();
    i_WETH.safeTransfer(_host, i_WETH.balanceOf(address(this)));
    // No deadline check, can withdraw at any time
}
```

Recommended Mitigation: Add deadline check before allowing withdrawals:

```
function withdraw() external onlyHost {
    require(block.timestamp > deadline, "Cannot withdraw before
deadline");
    address _host = getHost();
    i_WETH.safeTransfer(_host, i_WETH.balanceOf(address(this)));
}
```

Medium Risk Findings

[M-01] Unsafe ETH Refund Implementation

Description: The `_refundETH` function uses the deprecated `transfer()` method which has a 2300 gas limit and updates state after the external call.

Impact: - Refunds could fail due to gas limitations - Violates Checks-Effects-Interactions pattern - Potential reentrancy risk

Proof of Concept:

```
function _refundETH(address payable _to) internal {
    uint256 refundValue = etherBalance[_to];
    _to.transfer(refundValue);      // Uses transfer instead of
call
    etherBalance[_to] = 0;          // State update after
external call
}
```

Recommended Mitigation:

```
function _refundETH(address payable _to) internal {
    uint256 refundValue = etherBalance[_to];
    etherBalance[_to] = 0;
    (bool success, ) = _to.call{value: refundValue}("");
    require(success, "ETH transfer failed");
}
```

[M-02] Unrestricted Participation Status Changes Enable Event Planning Disruption

Description: The changeParticipationStatus() function allows users to toggle their participation status without any restrictions or cooldown period. This lack of rate limiting enables malicious participants to rapidly change their status, which can disrupt event planning and potentially cause denial of service by constantly changing the participant count.

Impact: Event planning becomes unreliable as participant counts can be artificially manipulated Host cannot accurately plan for the event due to unstable participant numbers Increased gas costs for the contract due to excessive state changes Event management becomes challenging as participants can change status unlimited times Potential griefing attack vector where a malicious user could programmatically flip their status to disrupt planning

Vulnerable Code:

```
// Missing Access Control
function changeParticipationStatus() external {
        if (participant[msg.sender]) {
            participant[msg.sender] = false;
        } else if (!participant[msg.sender] && block.timestamp <=
deadline) {
            participant[msg.sender] = true;
        } else {
            revert BeyondDeadline();
        }
        emit ChangedParticipation(msg.sender,
participant[msg.sender]);
    }
```

Proof of Concept:

```
function testParticipationStatusManipulation() public {
    // Setup
    vm.startPrank(host);
```

```
    cd.setDeadline(7); // Set deadline to 7 days
    vm.stopPrank();

    // Initial deposit to become participant
    vm.startPrank(user1);
    wbtc.mint(user1, 1 ether);
    wbtc.approve(address(cd), 1 ether);
    cd.deposit(address(wbtc), 1 ether);

    // Demonstrate status manipulation
    for(uint256 i = 0; i < 100; i++) {
        cd.changeParticipationStatus(); // Toggle from true to
false
        assertEq(cd.getParticipationStatus(user1), false);

        cd.changeParticipationStatus(); // Toggle from false to
true
        assertEq(cd.getParticipationStatus(user1), true);

        // Log every 10th iteration to demonstrate the
manipulation
        if(i % 10 == 0) {
            console.log("Iteration:", i);
            console.log("Participation status changed twice");
        }
    }
    vm.stopPrank();
}
```

Recommended Mitigation:

Implement a cooldown period between status changesConsider implementing a maximum number of status changes per address.Add a small fee for changing status multiple times to discourage abuse

[M-03] Manipulatable Deadline Setting

Description: The `setDeadline` function never sets `deadlineSet` to true, allowing the host to repeatedly modify the deadline.

Impact: - Host can continuously extend the deadline - Users cannot reliably plan their participation - Could be used to prevent refunds indefinitely

Proof of Concept:

```
function setDeadline(uint256 _days) external onlyHost {
    if (deadlineSet) {
```

```
        revert DeadlineAlreadySet();
    } else {
        deadline = block.timestamp + _days * 1 days;
        // deadlineSet is never set to true
    }
}
```

Recommended Mitigation:

```
function setDeadline(uint256 _days) external onlyHost {
    if (deadlineSet) {
        revert DeadlineAlreadySet();
    }
    deadline = block.timestamp + _days * 1 days;
    deadlineSet = true;
    emit DeadlineSet(deadline);
}
```

Low Risk Findings

[L-01] Missing Minimum Deposit Requirement

Description: The contract allows deposits of any amount, including zero, which could be used for griefing attacks.

Impact: - Users can register with minimal amounts - Potential for spam registrations - Inefficient use of contract storage

Proof of Concept:

```
function deposit(address _token, uint256 _amount) external
beforeDeadline {
    // No minimum amount check
    if (participant[msg.sender]) {
        balances[msg.sender][_token] += _amount;
    }
}
```

Recommended Mitigation: Add minimum deposit requirement:

```
uint256 public constant MINIMUM_DEPOSIT = 1e15;

function deposit(address _token, uint256 _amount) external
beforeDeadline {
    require(_amount >= MINIMUM_DEPOSIT, "Deposit too small");
    // Rest of the function
}
```

[L-02] Inconsistent Participation Logic Between ETH and ERC20

Description: The `receive()` function automatically sets participation to true, while ERC20 deposits have a different flow.

Impact: - Inconsistent user experience - Potential confusion for participants - Different behavior for different asset types

Proof of Concept:

```
receive() external payable {
    etherBalance[msg.sender] += msg.value;
    emit NewSignup(msg.sender, msg.value, true);
    // Automatically considered participant
}
```

Recommended Mitigation: Standardize participation logic:

```
receive() external payable {
    if (!participant[msg.sender]) {
        participant[msg.sender] = true;
    }
    etherBalance[msg.sender] += msg.value;
    emit NewSignup(msg.sender, msg.value,
participant[msg.sender]);
}
```

Informational Findings

[I-01] Missing Zero Address Validation

Description: The contract lacks zero address validation in critical functions like `changeHost` and the constructor.

Impact: - Potential contract lockup if host is set to zero address - No recovery mechanism if zero address is set

Recommended Mitigation: Add zero address validation:

```
function changeHost(address _newHost) external onlyHost {
    require(_newHost != address(0), "Zero address not allowed");
    if (!participant[_newHost]) {
        revert OnlyParticipantsCanBeHost();
    }
    host = _newHost;
    emit NewHost(host);
}
```