Christmas Dinner First Flight Audit Report

Lead Auditors:

- Aditya Mishra

Table of Contents

About the Project

About

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.

We try to address the following problems in the oraganization of such events:

- **Funding Security**: Organizing a social event is tough, people often say "we will attend" but take forever to pay their share, with our Christmas Dinner Contract we directly "force" the attendees to pay upon signup, so the host can plan properly knowing the total budget after deadline.

- **Organization**: Through funding security hosts will have a way easier time to arrange the event which fits the given budget. No Backsies.

Actors

Actors:

- `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.

Disclaimer

The Aditya Mishra team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for 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.

Risk Classification

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope (contracts)

```
All Contracts in `src` are in scope.
```

```
src/
├── ChristmasDinner.sol
```

Compatibilities

```
Compatibilities:
  Blockchains:
      - Ethereum
  Tokens:
      - ETH
      - WETH
      - WBTC
      - USDC
```

High

[H-1] Critical Reentrancy Vulnerability in NonReentrant Modifier

Description: The nonReentrant modifier implementation contains a critical flaw in its reentrancy protection mechanism. The current implementation fails to set the lock before function execution and incorrectly manages the lock state. The actual vulnerability is found here:

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

Impact:

- Allows malicious contracts to reenter the contract during execution
- Potential complete drain of contract funds
- Bypassing of intended security measures

Proof of Concept:

```
// In test file
function testReentrancyAttack() public {
    address attacker = makeAddr("attacker");
    // Setup attacker contract that implements receive() to reenter
    ReentrancyAttacker attackerContract = new
ReentrancyAttacker(address(christmasDinner));

    // Fund attacker
    vm.deal(address(attackerContract), 1 ether);

    // Trigger attack
    attackerContract.attack{value: 1 ether}();
```

```
    // Verify multiple reentrant calls succeeded
    assertGt(attackerContract.callCount(), 1);
}

contract ReentrancyAttacker {
    ChristmasDinner target;
    uint256 public callCount;

    constructor(address _target) {
        target = ChristmasDinner(_target);
    }

    function attack() external payable {
        // Initial deposit
        (bool success,) = address(target).call{value: 1 ether}("");
        require(success, "Initial deposit failed");
    }

    receive() external payable {
        callCount++;
        if(callCount < 3) {
            target.refund();
        }
    }
}
```

Recommended Mitigation: Implement proper reentrancy guard:

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

[H-2] Missing Deadline Validation in `withdraw()` Function

Description: The `withdraw()` function lacks deadline validation, allowing the host to withdraw funds before the deadline. The actual vulnerability is found here:

```
function withdraw() external onlyHost {
    address _host = getHost();
    i_WETH.safeTransfer(_host, i_WETH.balanceOf(address(this)));
    i_WBTC.safeTransfer(_host, i_WBTC.balanceOf(address(this)));
    i_USDC.safeTransfer(_host, i_USDC.balanceOf(address(this)));
}
```

Impact:

- Host can drain all user deposits before the event
- Complete loss of user funds
- Breaks the trust model of the contract

Proof of Concept:

```
function testPrematureWithdraw() public {
    // Setup
    vm.prank(host);
    christmasDinner.setDeadline(7); // 7 days deadline

    // Make deposits
    vm.deal(alice, 1 ether);
    vm.prank(alice);
    (bool success,) = address(christmasDinner).call{value: 1 ether}("");
    require(success);

    // Host withdraws immediately
    vm.prank(host);
    christmasDinner.withdraw();

    // Verify premature withdrawal succeeded
    assertEq(address(christmasDinner).balance, 0);
}
```

Recommended Mitigation: Add deadline check to `withdraw()` function:

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

[H-3] Unsafe ETH Transfer Without Address Validation

Description: The `refund()` function in the ChristmasDinner contract sends ETH to users without performing proper address validation checks. This is particularly concerning in the internal `_refundETH` function that's called by `refund()`. The actual vulnerability will be found here in the contract:

```
function refund() external nonReentrant beforeDeadline {
    address payable _to = payable(msg.sender);
    _refundERC20(_to);
    _refundETH(_to);
    emit Refunded(msg.sender);
}
```

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

Impact:

- Potential Loss of Funds:
    1. If `msg.sender` is a contract address that can't handle ETH (no receive/fallback function)
    2. If `msg.sender` is an invalid or incorrectly formatted address
    3. If the address is self-destructed between balance check and transfer
- Failed Transactions:
    1. Transfer to invalid addresses will cause the entire transaction to revert
    2. This could block legitimate refund attempts
- Smart Contract Integration Issues:
    1. Contracts interacting with this function might fail unexpectedly
    2. No way to handle failed transfers gracefully

Proof of Concept:

```
// Test contract demonstrating the vulnerability
contract VulnerabilityTest {
    ChristmasDinner dinner;

    constructor(address _dinner) {
        dinner = ChristmasDinner(_dinner);
    }

    // This contract has no receive() or fallback()
    function depositAndRefund() external payable {
        // Deposit ETH
        (bool success,) = address(dinner).call{value: msg.value}("");
        require(success, "Deposit failed");

        // Attempt refund - This will fail because contract can't receive ETH
        dinner.refund();
    }
}

// Test script
function testFailedRefund() public {
    // Deploy vulnerable contract
    ChristmasDinner dinner = new ChristmasDinner(wbtc, weth, usdc);

    // Deploy test contract
    VulnerabilityTest test = new VulnerabilityTest(address(dinner));
```

```
    // Attempt deposit and refund
    vm.expectRevert(); // Expected to revert
    test.depositAndRefund{value: 1 ether}();
}
```

Recommended Mitigation:

1. Add Address Validation:

```
function refund() external nonReentrant beforeDeadline {
    address payable _to = payable(msg.sender);
    require(_to != address(0), "Invalid address");
    require(_to != address(this), "Cannot refund to contract");

    _refundERC20(_to);
    _refundETH(_to);
    emit Refunded(msg.sender);
}
```

2. Implement Safe Transfer Pattern:

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

    // Clear balance before transfer to prevent reentrancy
    etherBalance[_to] = 0;

    // Use low-level call with gas stipend
    (bool success, ) = _to.call{value: refundValue, gas: 2300}("");
    require(success, "ETH transfer failed");

    emit ETHRefunded(_to, refundValue);
}
```

3. Add Try-Catch Pattern:

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

    try _to.call{value: refundValue, gas: 2300}("") returns (bool success) {
        require(success, "ETH transfer failed");
        emit ETHRefunded(_to, refundValue);
    } catch {
        // Revert the balance change if transfer fails
        etherBalance[_to] = refundValue;
```

```
            emit RefundFailed(_to, refundValue);
            revert("ETH transfer failed");
        }
    }
```

Medium

[M-1] Unprotected Deadline Extension

Description: While `setDeadline` can only be called once, it allows setting an arbitrarily long deadline.

The actual vulnerability is found here:

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

Impact:

- Potential indefinite fund lock
- Trust exploitation
- User fund inaccessibility

Recommended Mitigation: Add a maximum deadline limit:

```
uint256 public constant MAX_DEADLINE_DAYS = 30;

function setDeadline(uint256 _days) external onlyHost {
    require(_days <= MAX_DEADLINE_DAYS, "Deadline too far in future");
    require(_days > 0, "Deadline must be future date");
    if(deadlineSet) {
        revert DeadlineAlreadySet();
    }
    deadline = block.timestamp + _days * 1 days;
    deadlineSet = true;
    emit DeadlineSet(deadline);
}
```

[M-2] Missing Zero-Address Validation

Description: The `constructor` doesn't validate if the token addresses are zero addresses. The actual vulnerability is found here:

```
constructor (address _WBTC, address _WETH, address _USDC) {
    host = msg.sender;
    i_WBTC = IERC20(_WBTC);
    whitelisted[_WBTC] = true;
    i_WETH = IERC20(_WETH);
    whitelisted[_WETH] = true;
    i_USDC = IERC20(_USDC);
    whitelisted[_USDC] = true;
}
```

Impact: Deploying with zero addresses for tokens would break core contract functionality.

Recommended Mitigation: Add zero-address checks:

```
constructor (address _WBTC, address _WETH, address _USDC) {
    require(_WBTC != address(0), "WBTC cannot be zero address");
    require(_WETH != address(0), "WETH cannot be zero address");
    require(_USDC != address(0), "USDC cannot be zero address");
    host = msg.sender;
    i_WBTC = IERC20(_WBTC);
    whitelisted[_WBTC] = true;
    i_WETH = IERC20(_WETH);
    whitelisted[_WETH] = true;
    i_USDC = IERC20(_USDC);
    whitelisted[_USDC] = true;
}
```

Low

[L-1] Missing Events for Critical Operations

Description: Some critical state changes don't emit events.

Impact: Harder to track contract state changes off-chain and potential issues with front-end synchronization.

Recommended Mitigation: Add events for:

- Deadline changes
- Token balance changes
- Participation status changes

[L-2] Inconsistent Balance Management

Description: The contract handles ETH and ERC20 balances differently (separate mappings).

Impact: Could lead to maintenance issues and potential bugs in future updates.

Recommended Mitigation: Unify the balance tracking system using a single mapping structure:

```
mapping(address user => mapping(address token => uint256 balance)) public
balances;
// Use address(0) for ETH
```

[L-3] Missing Contract Pause Mechanism

Description: No way to pause the contract in case of emergencies.

Impact: If a vulnerability is discovered, there's no way to prevent further deposits or withdrawals.

Recommended Mitigation: Implement OpenZeppelin's `Pausable` contract and add pause functionality for the host.

[L-4] Unsafe ERC20 Operations Used

Description: The contract uses unsafe ERC20 operations that may not behave as expected. Some ERC20 implementations have inconsistent return value behaviors.

```
_to.transfer(refundValue);
```

Impact:

- Potential silent failures in token transfers
- Inconsistent behavior across different token implementations
- Risk of failed transactions without proper error handling

Recommended Mitigation: Implement OpenZeppelin's SafeERC20 library for all ERC20 operations:

```
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

using SafeERC20 for IERC20;

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)]);
}
```

[L-5] Missing Zero-Address Validation

Description: The contract assigns values to address state variables without checking for the zero address (0x0).

```
host = _newHost;
```

Impact:

- Potential assignment of invalid addresses
- Risk of function calls to zero address
- Possible loss of contract control

Recommended Mitigation: Add zero-address validation before address assignments:

```
function changeHost(address _newHost) external onlyHost {
    require(_newHost != address(0), "Invalid zero address");
    if(!participant[_newHost]) {
        revert OnlyParticipantsCanBeHost();
    }
    host = _newHost;
    emit NewHost(host);
}
```

[L-6] Single-Use Modifier Implementation

Description: The `nonReentrant` modifier is only used once in the contract, which may not justify the gas cost of implementing it as a modifier.

```
modifier nonReentrant() {
```

Impact:

- Increased deployment gas costs
- Code complexity without proportional benefit
- Reduced code maintainability

Recommended Mitigation: Either:

1. Incorporate the modifier logic directly into the function:

```
function refund() external beforeDeadline {
    require(!locked, "No re-entrancy");
    locked = true;

    address payable _to = payable(msg.sender);
    _refundERC20(_to);
    _refundETH(_to);
    emit Refunded(msg.sender);
```

```
        locked = false;
    }
```

2. Or justify modifier usage by implementing it in multiple functions where reentrancy protection is needed.

[L-7] Non-Constant State Variable Declaration

Description: The `deadlineSet` state variable is initialized at deployment but could be declared as a constant to save gas.

```
bool public deadlineSet = false;
```

Impact:

- Unnecessary gas consumption
- Suboptimal contract efficiency
- Higher storage costs

Recommended Mitigation: For variables that never change post-deployment, use the `constant` keyword:

```
bool public constant INITIAL_DEADLINE_SET = false;
```

If the variable needs to be mutable, ensure this is documented and intentional.