[H-1] `ChristmasDinner::changeParticipationStatus()` has no balance checks, meaning non-funders can become participants

Summary

the function `changeParticipationStatus()`, is set as an external function with no deposited balance checks, meaning non-funders can become participants, which opposes the contract's logic as stated in the docs: `Participant: Attendees of the event which provided some sort of funding.`

Vulnerability Details

allowing non-funders to become participants will disrupt's the event's organization which can lead to users who did not provide any funding benefiting from the event without contributing, disrupting both the contract's intended logic and the host's control over participation.

Impact

anyone can become a participant, distorting the contract's intended functionality for overall event organization

Tools Used

Manual Review, Foundry

Recommendations

Add a balance check before changing the state of the participant

```
function changeParticipationStatus() external {
    if(participant[msg.sender]) {
        participant[msg.sender] = false;
    } else if(!participant[msg.sender] && block.timestamp <= deadline) {

+       bool hasBalance = etherBalance[msg.sender] > 0 ||
+       balances[msg.sender][address(i_WETH)] > 0 || balances[msg.sender]
[address(i_WBTC)] || balances[msg.sender][address(i_USDC) > 0];

+       if(!hasBalance) {
+           revert("you dont have deposit in contract");
+       }

        participant[msg.sender] = true;
    } else {
        revert BeyondDeadline();
    }
```

```
    emit ChangedParticipation(msg.sender, participant[msg.sender]);
    }
```

[H-2] Contract doesn't have a withdrawal function for native ETH, meaning host cannot access funds

Summary

The Christmas Dinner contract handles deposits for ether, but doesn't allow for withdrawals. meaning participants ETH funds will be locked in the contract forever which goes against the contract's intended design

Vulnerability Details

When ETH is sent to the contract, the host can't withdraw to collect ETH to plan for the event. meaning any ETH sent will be stuck without any purpose in the contract as opposed to the intended design of the contract which said `About: It is supposed to sign up participants for a social christmas dinner (or any other dinner), while collecting payments for signing up`.

Impact

Host cannot access funds for dinner

Tools Used

Manual Review, Foundry

Recommendations

add a withdraw function for ether

```
function withdrawEth() public onlyHost {
    uint256 amountToWithdraw = address(this).balance;
    (bool success,) = payable(host).call{value: amountToWithdraw}("");
    if(!success) {
        revert ("transfer failed");
    }
}
```

[H-3] Reentrancy vulnerability in `ChristmasDinner::refund()`

Summary

The `refund()` fucntion is vulnerable to Re-entrancy attacks by a malicious user. The vulnerability is caused by an incorrect implementation of the `nonReentrant` modifier and failure to follow the Checks-Effects-Interactions pattern.

Vulnerability Details

The `nonReentrant` modifier is implemented incorrectly and so state changes will occur after external calls, allowing for reentrancy. Specifically:

1. The modifier never sets `locked = true`, only checks and sets it to false:

```
modifier nonReentrant() {
    require(!locked, "No re-entrancy");
    _;              // Function executes with lock still false
    locked = false; // Lock is set to false after execution
}
```

2.The `refund()` function performs external calls before state changes:

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

3.The `_refundETH()` function makes external calls before updating state:

```
function _refundETH(address payable _to) internal {
    uint256 refundValue = etherBalance[_to];
    _to.transfer(refundValue);     // External call happens before state update
    etherBalance[_to] = 0;         // State is updated too late
}
```

This allows a malicious contract to:

1. Make an initial deposit to become a participant
2. Call `refund()`
3. When receiving ETH in the fallback function, call `refund()` again
4. Repeat step 3 until the contract is drained

Impact

A malicious user/contract can drain the entire ETH balance from our contract. This is a critical severity issue because

1. It allows unauthorized withdrawal of funds
2. Can drain 100% of the contract's ETH balance
3. Requires minimal upfront capital to execute
4. Can be executed by any participant

Tools Used

Slither, Remix IDE

Recommendations

1. Follow CEI pattern (do necessary checks first, change the state, then do the interaction)
2. Use Openzeppelin's Reentrancy Guard contract

[H-4]: Denial of service: Unrestricted Zero-Value ETH Refunds Leading to Gas Exhaustion and making contract unusable

Summary:

The `refund()` and internal `_refundETH()` function in the ChristmasDinner contract fails to validate refund eligibility, allowing addresses without a balance in the contract to waste gas from the contract without restrictions.

Vulnerability details

Both the internal `_refundETH()` and external `refund()` function does not check if the caller (msg.sender) is eligible for a refund. For example if their ETH balance is 0 they can unconditionally transfer 0 ETH based on the etherBalance mapping, which can waste a lot of gas. as the EVM allows for 0 value ETH transfers.

Impact

When a malicious user without a balance keeps calling the contract's `refund()` function, the contract will keep sending 0 ether and can eventually run out of gas. Making the entire contract unusable

Tools Used

Manual Review, Foundry

Recommendations

in the `refund()` function add a check to ensure the refundee's ETH balance is greater than 0

```
    function refund() external nonReentrant beforeDeadline {
+       require(etherBalance[msg.sender] > 0, "no ether balance in contract");
        address payable _to = payable(msg.sender);
        _refundERC20(_to);
```

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

[M-1] Deadline can still be updated even after it has been set

Summary

The `setDeadline()` function can still be called even after deadline has been set due to an incorrect logic error, and going against the function's intended design

Vulnerability Details

The function is not updating the `deadlineStatus` to true after it has been set, making room to extend or shorten deadlines.

Impact

Unpredictable behavior or malicious misuse by the host, such as extending the deadline indefinitely to allow or prevent additional participants.

Tools Used

Manual Review, Foundry

Recommendations

we update the `deadlineStatus` after the function is called

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