First Flight N31 Christmas Dinner

Prepared by: 0xKoiner

First Flight N31 Christmas Dinner

Prepared by: 0xKoiner Lead Auditors: 0xKoiner

- [0xKoiner] (#https://twitter.com/0xKoiner)

Assisting Auditors:

- None

Table of contents

▶ Details

See table

About OxKoiner

Hi, I'm OxKoiner, a developer and smart contract auditor based in sunny Barcelona. My journey in the crypto industry started back in 2016, and it's been an exciting ride ever since. With a background in Computer Science from the Technion (Israel Institute of Technology), I've built up a versatile skill set that spans Python development and smart contract security.

My Experience Python Development I have a solid background in Python, focusing on backend development, software automation, testing, and Web3 libraries. I enjoy building tools and solutions that help simplify complex problems, and I'm always looking for new ways to streamline workflows and processes.

Solidity & Smart Contract Development I'm a junior Solidity developer with a growing passion for low-level aspects of Ethereum development, including OpCodes, Huff, and Assembly. I love diving into the details of smart contract design, aiming to write clean, efficient, and secure code. I'm still learning every day, but I enjoy the challenge of constantly improving.

Auditing & Security I've completed several courses focused on smart contract audits and enjoy the process of analyzing code for potential vulnerabilities. My toolkit includes:

Foundry & HardHat: For testing and development. Slither & Aderyn: For static analysis and finding common issues. Certora: For formal verification to ensure contract safety. I take a careful, methodical approach when auditing, trying to catch even the smallest issues that could become big problems. I believe every audit is a chance to learn and help make the Web3 space a bit safer.

My Approach I'm motivated by a genuine interest in blockchain technology and its potential to bring about positive change. I'm not an expert in everything, but I'm always eager to learn and collaborate with others. If you're working on an interesting project or need a fresh set of eyes for an audit, I'd love to connect and see how I can help.

Let's build something great together!

Disclaimer

The OxKoiner team has made every effort to identify potential vulnerabilities within the time allocated for this audit. However, we do not assume responsibility for the findings or any issues that may arise after the audit. This security audit is not an endorsement of the project's business model, product, or team. The audit

was time-limited and focused exclusively on assessing the security of the Solidity code implementation. It is strongly recommended that additional testing and security measures be conducted by the project team.

Risk Classification

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

Scope

```
All Contracts in `src` are in 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. We try to address the following problems in the oraganization of such events, such as funding security and overall organization.

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	2

Severity	Number of issues found
Medium	3
Low	5
Total	10

Findings

High

[H-1] Flawed Implementation of nonReentrant Modifier

_Submitted by 0xKoiner

Summary

The `nonReentrant` modifier is intended to prevent reentrancy attacks by using a `locked` flag. However, the implementation has a logic flaw where the `locked` flag is only set to `false` after the function body executes. This flaw could allow reentrancy if `locked` is not properly set to `true` before executing the function body. Additionally, the `refund()` function does not fully leverage the intended reentrancy protection, as `locked = true` is not set before executing its logic.

Vulnerability Details

Current `nonReentrant` Modifier Implementation:

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

Issues:

1. **Initial `locked` Value Logic:**
    - The modifier checks `require(!locked, "No re-entrancy");`. If `locked = false`, the check passes, which is expected. However, `locked` should immediately be set to `true` before entering the function body to ensure no other calls can pass the `require()` condition.
    - This implementation creates a small window where reentrancy might still be possible before `locked = true`is manually set in the function.
2. **Missing `locked = true` in Function Logic:**

- In the `refund()` function:

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

- The function relies on the `nonReentrant` modifier, but since `locked` is not set to `true` in the modifier before entering the function body, it does not fully protect against reentrancy.

3. **Potential Misuse:**
    - If developers rely on the current modifier implementation, they might assume reentrancy protection is active throughout the function body, which is not true without explicitly setting `locked = true` within the modifier.

Impact

- The current `nonReentrant` implementation is flawed and could potentially allow reentrancy if misused or combined with external calls.
- Functions relying on `nonReentrant` might be susceptible to reentrancy attacks unless additional protection logic is manually implemented.

Tools Used

- Manual code review.
- Analysis of `nonReentrant` pattern and Solidity best practices.

Recommendations

1. **Fix the `nonReentrant` Modifier Logic:** Update the `nonReentrant` modifier to set `locked = true` before the function body is executed. This ensures reentrancy protection is active during the entire execution of the function.

```
modifier nonReentrant() {
    require(!locked, "No re-entrancy");
+   locked = true; // Set locked to true immediately
    _;
    locked = false; // Reset locked after function execution
}
```

2. **Update the `refund()` Function:** With the corrected modifier, the `refund()` function will be properly protected. The corrected `nonReentrant` modifier eliminates the need to manually set `locked = true` inside the function body:

```
function refund() external nonReentrant beforeDeadline {
    address payable _to = payable(msg.sender);
```

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

3. **Review All Functions Using `nonReentrant`:** Ensure that all functions relying on the `nonReentrant` modifier are updated to follow the corrected implementation.4
4. Since solidty 0.8.24 its allowed design contracts with Transient Storage and be used for Reentrancy Locks modifier pattern. Its more gas efficient and saving slots in storage (priced at 100 gas.). Still the syntax only on low-level lang. and might be more difficult to implement into the basecode.

```
modifier nonReentrant() {
        assembly {
            if tload(0) { revert(0, 0) }
            tstore(0, 1)
        }
        _;
        assembly {
            tstore(0, 0)
        }
    }
```

Conclusion

The current `nonReentrant` modifier contains a logic flaw that reduces its effectiveness. Correcting the modifier to set `locked = true` before entering the function body will ensure proper reentrancy protection. The `refund()` function and others relying on this modifier will then be safeguarded against reentrancy attacks.

[H-2] Missing Ether Withdrawal in ChristmasDinner.sol Contract

_Submitted by 0xKoiner

Summary

The `withdraw()` function in the `ChristmasDinner` contract does not include logic for withdrawing Ether (ETH). This omission can lead to Ether being stacked indefinitely in the contract, potentially causing financial losses for users and impacting the usability of the contract.

Vulnerability Details

The `withdraw()` function in the `ChristmasDinner` contract only handles the transfer of ERC20 tokens back to the host's address. It does not include any logic for handling Ether deposits. As a result, users who deposit Ether into the contract and later wish to withdraw it cannot do so. This could lead to unexpected accumulation of Ether within the contract, making it inaccessible to users and posing a security risk.

Here is the relevant part of the code that requires modification:

```
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

- **Critical Risk**: Users who deposit Ether into the contract and wish to withdraw it are unable to do so. This could lead to financial losses and inconvenience for those users.
- **Inaccessibility of Funds**: Ether deposits will remain in the contract indefinitely, causing potential issues such as blocked funds and inability to meet withdrawal requests.
- **Security Concern**: Accumulation of Ether without withdrawal mechanisms could lead to vulnerabilities in the contract, impacting its overall security and reliability.

Tools Used

- **Manual Code Review**: The vulnerability was identified through a detailed code review of the `withdraw()` function and its interaction with Ether.
- **Solidity Best Practices Analysis**: Reviewing the contract against best practices for handling Ether deposits and withdrawals.

Recommendations

To mitigate the issue, update the `withdraw()` function to include the ability to transfer Ether from the contract to the host's address:

```
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)));
+   payable(_host).call{value: address(this).balance};
}
```

This modification will ensure that the contract can handle both ERC20 token withdrawals and Ether withdrawals, thus preventing the indefinite accumulation of Ether in the contract and ensuring better user experience and security.

Medium

[M-1]. Violation of the Correct Order of State Changes /CEI Pattern/ in _refundERC20 and _refundETH Functions

_Submitted by 0xKoiner

Summary

The functions `_refundERC20` and `_refundETH` in the `ChristmasDinner` contract do not follow the correct pattern of state changes (CEI - Consistency, Error-Checking, and Interaction). This violation could lead to unexpected behaviors and vulnerabilities, such as manipulation of contract state or unintended consequences in a reentrant call.

Vulnerability Details

Issue in `_refundERC20` Function

The `_refundERC20` function transfers the user's ERC20 balances first and then resets those balances to zero in the state variables. The function does not follow the CEI pattern, which could lead to race conditions or other unexpected issues:

```
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;
}
```

- **Violation of CEI Pattern**:
- The state changes should be separated into three distinct steps:

1. **Consistency**: Perform any error-checking operations to ensure the function preconditions are met (e.g., checking that balances are not zero before transferring).
2. **Error-Checking**: Perform state changes to the contract state or updates to variables.
3. **Interaction**: Perform the state-changing interactions (transfers in this case).

- In the current implementation, the state changes are not correctly ordered, which can lead to the following problems:
- **Race Conditions**: If the transfer of tokens fails (for example, due to insufficient funds), the state change will still occur, leaving the contract in an inconsistent state.
- **Double Withdrawals**: In case of reentrant calls, the state might allow a double withdrawal, as the balance changes are performed after the actual transfers.

Issue in `_refundETH` Function

The `_refundETH` function exhibits the same pattern of inconsistent state changes:

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

```
    etherBalance[_to] = 0;
}
```

- **Violation of CEI Pattern**:
- The state change order is incorrect because the `etherBalance[_to]` is reset before the transfer occurs.
- This can lead to race conditions where a reentrant call could manipulate the state between the reading of `etherBalance[_to]` and the subsequent `transfer`.

Impact

- **Unexpected Behavior**: The lack of CEI pattern compliance can lead to unexpected behaviors such as reentrancy vulnerabilities, double withdrawals, and state inconsistencies.
- **Reentrancy Vulnerability**: If the refund functions are called in a reentrant manner, the contract's state may allow for multiple transfers when it should only allow one.
- **Manipulation of State**: An attacker could potentially exploit the order of state changes to manipulate balances and withdraw funds more than once.

Tools Used

- Manual code review.
- Analysis of best practices in Solidity development and the correct implementation of the CEI pattern. Recommendations

Recommendations

1. **Follow the CEI Pattern**: Ensure that the functions `_refundERC20` and `_refundETH` follow the correct order of state changes:

- **Consistency**: Check that the balances are not zero or that they have a non-zero amount before transferring.
- **Error-Checking**: Check any additional conditions to prevent incorrect state changes.
- **Interaction**: Perform the transfer of funds.

2. **Revised Implementations**:

- For `_refundERC20`:

```
function _refundERC20(address _to) internal {
        uint256 amountWethToRefund = balances[_to][address(i_WETH)];
        uint256 amountWbtcToRefund = balances[_to][address(i_WBTC)];
        uint256 amountUsdcToRefund = 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, amountWethToRefund);
```

```
        i_WBTC.safeTransfer(_to, amountWbtcToRefund);
        i_USDC.safeTransfer(_to, amountUsdcToRefund);
    }
```

- For `_refundETH`:

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

3. **Review and Test**: Conduct tests and ensure that the corrected functions are properly handling reentrancy scenarios and are robust against common vulnerabilities.

Conclusion

The violation of the CEI pattern in the `_refundERC20` and `_refundETH` functions in the `ChristmasDinner` contract can lead to unexpected behaviors and vulnerabilities. Following the correct pattern will enhance the security and reliability of the contract by preventing manipulation and reentrancy attacks.

[M-2] Lack of Participation Status Update in receive() Function

_Submitted by 0xKoiner

Summary

The `receive()` function in the `ChristmasDinner` smart contract allows users to send Ether to the contract. However, it does not properly update the participant status for users, leading to potential issues where users can deposit Ether multiple times without being considered participants. Unlike ERC20 deposits, where participation status is automatically updated, Ether depositors must manually invoke the `changeParticipationStatus()` function to attend the event.

Vulnerability Details

The `receive()` function handles Ether deposits from users by updating the `etherBalance` mapping, but it does not update the `participant` status. The function simply adds the incoming Ether to the user's balance (`etherBalance[msg.sender] += msg.value`) and emits a `NewSignup` event. Unlike the `deposit` function for ERC20 tokens, which correctly checks the user's participation status and updates it accordingly, this function does not enforce participation status, allowing users to deposit Ether multiple times without being marked as participants.

This oversight can lead to the following vulnerabilities:

1. **Bypassing Participation Rules**: Users can deposit multiple times without being considered participants, potentially leading to misuse or abuse of the contract's features.

2. **Inconsistency**: The contract can become inconsistent in tracking who is actually participating in the event, which may lead to disputes or unexpected behaviors during the event.
3. **Manual Action Requirement for Ether Depositors**: Unlike ERC20 depositors who are automatically marked as participants, Ether depositors must manually invoke the `changeParticipationStatus()` function to attend the event. This creates an unnecessary barrier and adds complexity to user interactions with the contract.

Impact

- **Data Inconsistency**: Without proper status updates, the contract may not accurately reflect user participation.
- **Increased Risk of Abuse**: Users can deposit Ether without triggering the participant flag, circumventing participation rules.
- **Potential Exploitation**: Malicious actors could deposit Ether repeatedly, receiving event benefits without being registered as participants.
- **Manual Participation for Ether Depositors**: Ether depositors must actively manage their participation status, unlike ERC20 depositors.

Tools Used

- **Manual Code Review**: Reviewing contract logic and identifying discrepancies between functions handling different assets (Ether vs ERC20 tokens).
- **Static Analysis Tools**: Used to validate function logic and identify non-conforming behavior.

Recommendations

1. **Update the `receive()` function to include participation status**: Ensure that when Ether is received, the contract also sets the user's participation status to `true`. This can be done by modifying the `participant[msg.sender] = true;` line after updating the `etherBalance`.

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

2. **Code Consistency**: Maintain consistency between the handling of different asset deposits (Ether and ERC20 tokens) to avoid discrepancies in tracking user status.

3. **Additional Testing**: Rigorously test the contract's behavior with both ERC20 and Ether deposits to ensure all cases are covered, especially edge cases involving re-entrancy and multiple deposits.

4. **Simplify Ether Depositor Participation**: Consider simplifying the participation process for Ether depositors by automatically marking them as participants upon their deposit, similar to ERC20 depositors.

[M-3] Use call instead of transfer in _refundETH function to prevent ETH stack in the contract

_Submitted by 0xKoiner

Summary

The `_refundETH` function in the `ChristmasDinner` contract uses the `transfer` method to send Ether. Using `transfer` is unsafe due to its fixed gas limit, which may not be sufficient to complete more complex operations in the receiving contract, potentially leading to failure and loss of funds. This function should be modified to use the `call` method instead.

Vulnerability Details

In the `ChristmasDinner` contract, the `_refundETH` function:

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

uses the `transfer` method to send Ether to the recipient. The `transfer` function forwards only 2300 gas, which may not be enough to complete the transfer if the recipient's fallback function consumes more gas. This can lead to the contract's Ether balance being permanently stuck. Furthermore, if the `transfer` fails, it throws an exception, which makes handling errors difficult.

Using `transfer` does not provide feedback about whether the operation was successful; it either succeeds or throws an error. This makes it harder to handle errors gracefully in the contract.

Impact

Using `transfer` in the `_refundETH` function can lead to the following issues:

1. **Potential loss of Ether**: If the recipient's contract consumes more gas than provided by `transfer`, the transaction fails, and the Ether is lost.
2. **Unreachable Ether**: Ether can become permanently stuck in the contract, making it unavailable for withdrawals or refunds.
3. **Difficult error handling**: In cases where the `transfer` fails, it throws an error without providing information about the failure, complicating debugging and recovery efforts.

Tools Used

- Solidity compiler version 0.8.27
- Etherscan for contract analysis

Recommendations

To prevent issues with the `_refundETH` function, it is recommended to use the `call` method instead of `transfer`. The `call` method allows specifying the gas limit, which provides more control over the transaction execution. Here's how the function can be modified:

```
function _refundETH(address payable _to) internal {
    uint256 refundValue = etherBalance[_to];
-   _to.transfer(refundValue);
+   (bool success, ) = _to.call{value: refundValue}("");
+   require(success, "Failed to send Ether");
    etherBalance[_to] = 0;
}
```

By using `call`, you can specify a gas limit (such as `gasleft()` or a fixed number like `5000`) to ensure the recipient's contract can handle the transaction. This approach also returns a boolean value indicating success or failure, making error handling simpler and more predictable.

Low

[L-1] Improper Event Parameter Naming in Solidity Contracts

_Submitted by 0xKoiner

Summary

The Solidity contract defines multiple events with unnamed parameters. While functional, this approach can lead to readability issues, tooling incompatibility, and difficulties in off-chain event decoding. Proper naming of parameters in event declarations is essential for clarity and maintaining best practices in Solidity development.

Vulnerability Details

The following events are declared without parameter names:

```
event NewHost(address indexed);
event NewSignup(address indexed, uint256 indexed, bool indexed);
event GenerousAdditionalContribution(address indexed, uint256 indexed);
event ChangedParticipation(address indexed, bool indexed);
event Refunded(address indexed);
event DeadlineSet(uint256 indexed);
```

The parameters are correctly marked as `indexed` where required, which allows filtering of logs. However, the omission of parameter names introduces the following issues:

1. **ABI Generation:** Unnamed parameters are represented as empty strings (`""`) in the ABI, reducing its usability for event parsing and debugging.
2. **Readability:** Future developers may find it challenging to understand the purpose of each parameter without proper names.
3. **Tooling Compatibility:** Some tools, such as off-chain log parsers or SDKs (e.g., `ethers.js`), may encounter issues when interacting with unnamed event parameters.

ABI:


```
[
  {
    "inputs": [],
    "name": "BeyondDeadline",
    "type": "error"
  },
  {
    "inputs": [],
    "name": "DeadlineAlreadySet",
    "type": "error"
  },
  {
    "inputs": [],
    "name": "NotHost",
    "type": "error"
  },
  {
    "inputs": [],
    "name": "NotSupportedToken",
    "type": "error"
  },
  {
    "inputs": [],
    "name": "OnlyParticipantsCanBeHost",
    "type": "error"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "internalType": "address",
        "name": "",
        "type": "address"
      },
      {
        "indexed": true,
        "internalType": "bool",
        "name": "",
        "type": "bool"
      }
    ],
    "name": "ChangedParticipation",
    "type": "event"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "internalType": "uint256",
```

```
        "name": "",
        "type": "uint256"
      }
    ],
    "name": "DeadlineSet",
    "type": "event"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "internalType": "address",
        "name": "",
        "type": "address"
      },
      {
        "indexed": true,
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
      }
    ],
    "name": "GenerousAdditionalContribution",
    "type": "event"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "internalType": "address",
        "name": "",
        "type": "address"
      }
    ],
    "name": "NewHost",
    "type": "event"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "internalType": "address",
        "name": "",
        "type": "address"
      },
      {
        "indexed": true,
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
      },
```

```
      {
        "indexed": true,
        "internalType": "bool",
        "name": "",
        "type": "bool"
      }
    ],
    "name": "NewSignup",
    "type": "event"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "internalType": "address",
        "name": "",
        "type": "address"
      }
    ],
    "name": "Refunded",
    "type": "event"
  }
]
```


Impact

While the contract is functional, the lack of descriptive names in event parameters:

- Reduces maintainability and readability of the contract.
- Makes it harder to decode events using automated tools, impacting off-chain integration.
- Risks potential misinterpretation of the contract's intent and structure.

Tools Used

- Manual code review.
- Solidity compiler for verification.
- Foundry for event testing and log inspection.

Recommendations

To address the issues, parameter names should be added to all events. For example, update the events as follows:

```
event NewHost(address indexed _addr);
event NewSignup(address indexed _addr, uint256 indexed _value, bool
indexed _status);
event GenerousAdditionalContribution(address indexed _addr, uint256
indexed _amount);
event ChangedParticipation(address indexed _addr, bool indexed _isActive);
```

```
event Refunded(address indexed _addr);
event DeadlineSet(uint256 indexed _timestamp);
```

This update will:

- Improve the contract's readability and maintainability.
- Ensure compatibility with off-chain tools and SDKs.
- Provide clarity for future developers working with the contract.

[L-2] Redundant Getter Function for Public State Variable

_Submitted by 0xKoiner

Summary

The contract defines a public state variable `host` as `address public host;`, which automatically provides a getter function for accessing its value. However, a separate explicit getter function `getHost()` is also implemented, introducing unnecessary redundancy. This redundant getter is used in the `withdraw()` function, where directly accessing the state variable `host` would be more efficient and clear.

Vulnerability Details

The code includes the following redundant getter:

```
function getHost() public view returns (address _host) {
    return host;
}
```

Since `host` is already declared as `public`, the Solidity compiler automatically generates a getter function with the same functionality. The explicit implementation of `getHost()` adds no value and introduces unnecessary complexity.

In the `withdraw()` function:

```
function withdraw() external onlyHost {
    address _host = getHost(); // Uses redundant getter
    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)));
}
```

The use of `getHost()` adds overhead and decreases readability. Instead, the variable `host` can be accessed directly:

```
address _host = host;
```

Impact

The explicit getter introduces:

1. **Redundancy:** Public state variables already have built-in getter functions, so implementing an additional one is unnecessary.
2. **Code Complexity:** Using the redundant getter in other functions (e.g., `withdraw()`) increases the complexity and makes the code less intuitive.
3. **Gas Overhead:** While minimal, calling an external getter adds slightly more gas cost than directly accessing the state variable.

Tools Used

- Manual code review.
- Solidity compiler analysis for behavior confirmation.

Recommendations

1. **Remove the Redundant Getter:** Eliminate the `getHost()` function since it duplicates the functionality provided by the `public` state variable.

```
// Remove this function
function getHost() public view returns (address _host) {
    return host;
}
```

2. **Directly Access the State Variable:** In the `withdraw()` function and other parts of the code, replace `getHost()`with direct access to the `host` variable:

```
function withdraw() external onlyHost {
    address _host = host; // Directly access the state variable
    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)));
}
```

These changes will improve code clarity, reduce redundancy, and ensure better gas efficiency.

[L-3] Internal visibility for state variables prevents user transparency

_Submitted by 0xKoiner

Summary

The `ChristmasDinner` contract uses `internal` visibility for several key state variables, including participant status, Ether deposits, and token balances. As a result, users cannot access their information directly on-chain to verify their participation status, deposit amounts, or token balances. Making these state variables `public` or implementing `getter`functions would allow users to query their status and balances, improving transparency and user trust.

Vulnerability Details

The following state variables in the contract have `internal` visibility:

```
mapping(address user => bool) participant;
mapping(address token => bool) whitelisted;
mapping(address user => uint256 amount) etherBalance;
mapping(address user => mapping(address token => uint256 balance))
balances;
```

Currently, users cannot query these values directly, as `internal` visibility restricts access to within the contract or derived contracts. Without transparency, users must rely on off-chain data or third-party services to track their balances and statuses, which increases the risk of misinformation or errors.

Impact

1. **Lack of transparency**: Users cannot verify their own data on-chain, leading to reduced trust in the contract.
2. **Increased support workload**: Users might frequently request information from the contract owner or developer team, adding unnecessary overhead.
3. **Reduced user experience**: Lack of accessible data makes the contract less user-friendly and might deter potential users.

Tools Used

- Solidity compiler version 0.8.27
- Manual contract review

Recommendations

To improve transparency and user experience, the contract should expose the state variables as `public` or implement dedicated `getter` functions.

Option 1: Make variables `public`

Modify the visibility of the state variables as follows:

```
- mapping(address user => bool) participant;
- mapping(address token => bool) whitelisted;
- mapping(address user => uint256 amount) etherBalance;
- (address user => mapping(address token => uint256 balance)) balances;
```

```
+ mapping(address user => bool) public participant;
+ mapping(address token => bool) public whitelisted;
+ mapping(address user => uint256 amount) public etherBalance;
+ mapping(address user => mapping(address token => uint256 balance))
public balances;
```

This approach automatically generates getter functions for each variable, allowing users to query their status and balances on-chain directly.

Option 2: Add custom getter functions

Alternatively, implement explicit getter functions:

```
function isParticipant(address user) public view returns (bool) {
    return participant[user];
}

function isTokenWhitelisted(address token) public view returns (bool) {
    return whitelisted[token];
}

function getEtherBalance(address user) public view returns (uint256) {
    return etherBalance[user];
}

function getTokenBalance(address user, address token) public view returns
(uint256) {
    return balances[user][token];
}
```

This approach provides more flexibility if you need additional logic or customization in the getter functions.

Recommendations Summary

- Use the `public` visibility for the state variables to automatically generate getter functions.
- Alternatively, implement explicit getter functions for more control and customization.

Exposing these variables improves user experience, transparency, and trust in the contract.

[L-4] Lack of Validation for Zero Deposit Amount in deposit Function

_Submitted by 0xKoiner

Summary

The `deposit` function lacks validation for a deposit amount of zero, allowing users to participate in the event and emit the `NewSignup` event without actually transferring any funds. This bypasses the intended logic of requiring a monetary commitment to participate in the event.

Vulnerability Details

Code Location

```
function deposit(address _token, uint256 _amount) external beforeDeadline
{
    if (!whitelisted[_token]) {
        revert NotSupportedToken();
    }
    if (participant[msg.sender]) {
        balances[msg.sender][_token] += _amount;
        IERC20(_token).safeTransferFrom(msg.sender, address(this),
_amount);
        emit GenerousAdditionalContribution(msg.sender, _amount);
    } else {
        participant[msg.sender] = true;
        balances[msg.sender][_token] += _amount;
        IERC20(_token).safeTransferFrom(msg.sender, address(this),
_amount);
        emit NewSignup(
            msg.sender,
            _amount,
            getParticipationStatus(msg.sender)
        );
    }
}
```

Description

The function does not check whether the `_amount` is greater than zero before allowing a user to participate. This means a user can:

1. Call the `deposit` function with `_amount = 0`.
2. Bypass the monetary commitment expected for participation.
3. Trigger the `NewSignup` event, registering as a participant without transferring any funds.

This vulnerability undermines the integrity of the participation logic and could lead to abuse, as users can become participants for free.

Comparison with `changeParticipationStatus`

The `changeParticipationStatus` function properly toggles participation status but does not allow users to make financial commitments. By exploiting the `deposit` function with `_amount = 0`, a user can achieve similar results with the added benefit of misleadingly registering as a paying participant.

Impact

This vulnerability allows malicious or careless users to:

1. Register as event participants without transferring any funds.

2. Trigger misleading events (`NewSignup`) without any financial commitment.
3. Potentially disrupt event organizers who rely on deposit amounts to gauge event funding or capacity.

Tools Used

- Manual code review
- Static analysis

Recommendations

1. **Add Validation for Non-Zero Amounts**: Ensure `_amount > 0` in the `deposit` function before proceeding.

```
if (_amount == 0) {
    revert ZeroDepositNotAllowed();
}
```

2. **Emit Separate Event for Zero Contributions**: If zero contributions are intentionally allowed, emit a different event to distinguish them from actual deposits.

```
if (_amount == 0) {
    emit ZeroContributionAttempt(msg.sender);
    return;
}
```

3. **Strengthen Event Logic**: Refactor the `deposit` function to ensure that only valid deposits trigger `NewSignup` or `GenerousAdditionalContribution` events.

4. **Test Cases**: Include test cases for zero deposit scenarios to ensure proper validation and behavior.

By implementing these measures, the contract can ensure that participation and event signup logic remains robust and secure.

[L-5] Lack of Access Control in Refund Function

_Submitted by 0xKoiner

Summary

The `refund` function in the contract lacks proper access control, allowing any user, regardless of whether they have deposited assets or not, to call the function and trigger refund processes. This can lead to unintended behavior and potential abuse. Adding a proper validation check for deposited assets before processing refunds is necessary to secure the function.

Vulnerability Details

The current implementation of the `refund` function does not verify whether the caller has deposited any assets before allowing the refund process. As a result, users who have not deposited any assets can still call the function, leading to redundant operations and unnecessary state changes. The code snippet below illustrates the issue:

```
function refund() external nonReentrant beforeDeadline {
    /// @audit = everyone can call refund function // lack of access
    address payable _to = payable(msg.sender);
    _refundERC20(_to);
    _refundETH(_to);
    emit Refunded(msg.sender);
}
```

This lack of a proper condition to validate the presence of refundable assets can allow abuse and affect the contract's efficiency and gas usage.

Impact

- Unnecessary operations for users with no refundable assets.
- Potential confusion among users when calling the refund function without prior deposits.
- Increased gas costs due to redundant operations.

Tools Used

Manual code review.

Recommendations

Introduce a validation check in the `refund` function to ensure that the caller has refundable assets before processing the refund. The following code snippet demonstrates the corrected implementation:

```
function refund() external nonReentrant beforeDeadline {
    address payable _to = payable(msg.sender);

    // Check if the user has any refundable assets
    if (
        balances[_to][address(i_USDC)] == 0 &&
        balances[_to][address(i_WBTC)] == 0 &&
        balances[_to][address(i_WETH)] == 0 &&
        etherBalance[_to] == 0
    ) {
        revert("No Assets to Refund");
    }

    // Proceed with the refund
    _refundERC20(_to);
    _refundETH(_to);
    emit Refunded(msg.sender);
}
```

Explanation of the Condition

- `balances[_to][i_USDC] == 0`: Checks if the user has no USDC balance.
- `balances[_to][i_WBTC] == 0`: Checks if the user has no WBTC balance.
- `balances[_to][i_WETH] == 0`: Checks if the user has no WETH balance.
- `etherBalance[_to] == 0`: Checks if the user has no ETH balance.

If all these conditions are true, the function will revert with the message "No Assets to Refund."

Recommendations Summary

1. Add a validation check to ensure the caller has refundable assets.
2. Use a comprehensive condition to check all supported asset balances.
3. Revert the transaction if the user has no refundable assets to prevent unnecessary operations.

Implementing these recommendations will enhance the function's security and operational efficiency.