Security Audit Report for CrowdFunding Smart Contract

Auditor: Mahdifa
Date: Semtember 14, 2025
Contact: karizmeh811@proton.me | GitHub: mahdifa811

Abstract

This report presents findings from a comprehensive security audit of the `CrowdFunding.sol` smart contract, conducted voluntarily to contribute to the blockchain community and enhance my auditing portfolio. I identified 9 issues in this contract, ranging from critical vulnerabilities to informational improvements. Detailed recommendations are provided to ensure the contract's security and reliability.

Scope

- **Contract**: `CrowdFunding.sol`
- **Version**: `pragma solidity ^0.8.19`
- **Libraries**: OpenZeppelin `Ownable`, `SafeMath`
- **Functions Audited**: All public and internal functions, including `createCampaign`, `donateToCampaign`, `payOutToCampaignTeam`, `deleteCampaign`, `haltCampaign`, `updateCampaign`, `getCampaigns`, `_calculateTax`, `withdrawFunds`, and `_refundDonators`.

Methodology

The audit employed a rigorous methodology to ensure thorough coverage:

- **Manual Code Review**: Line-by-line analysis to identify logical flaws, access control issues, and vulnerabilities.
- **Dynamic Testing**: Simulated attack scenarios to validate findings.
- **Standards Compliance**: Aligned with Smart Contract Weakness Classification (SWC) and Ethereum security best practices.

Executive Summary

The audit identified 9 issues:

- **3 High-severity vulnerabilities**: Related to fund management and campaign ID integrity.
- **2 Medium-severity issues**: Logical errors in donation limits and emergency refund logic.
- **3 Low-severity issues**: In event emission, tax calculation, and campaign halt bypassing.
- **1 Informational issue**: In data retrieval.

Each finding includes a detailed description, impact, proof of concept, and mitigation recommendations. Addressing High-severity issues is critical to prevent financial loss and data corruption.

Findings

Finding 1: Reentrancy Vulnerability in Refund Process

Severity: High
Status: Open

Description: The `_refundDonators` function, invoked by `deleteCampaign`, updates `amountCollected` after external calls to `_payTo`. This allows a malicious campaign owner to reenter `payOutToCampaignTeam` during the refund process, enabling double-spending of campaign funds (refund + payout minus tax).

Impact:

- Drains funds from the contract, affecting other campaigns.
- Undermines platform trust due to financial losses.

Prerequisites:

- Attacker deploys a malicious contract that serves as the campaign owner.
- The malicious contract has a `receive` or `fallback` function to perform reentrancy.
- Campaign has non-zero `amountCollected`.
- The campaign deadline must have passed (`block.timestamp >= campaign.deadline`).

Proof of Concept:

1. Attacker deploys a malicious contract with a `receive` function that calls `payOutToCampaignTeam(ID)`.

2. Using the malicious contract, attacker calls `createCampaign` with `_owner` set to the address of the malicious contract (ensuring the malicious contract is the campaign owner).
3. Malicious contract donates 1 ETH to its own campaign (ID 1).
4. Malicious contract calls `deleteCampaign(1)`, triggering `_refundDonators`.
5. During `_payTo` to the malicious contract (as the donor), it reenters `payOutToCampaignTeam(1)`. The reentrancy succeeds because `msg.sender` (malicious contract) matches the campaign `owner`, passing the `privilageEntity` modifier.
6. Attacker receives 1 ETH (refund) + 0.9 ETH (payout after 10% tax), totaling 1.9 ETH from 1 ETH donated, draining contract funds.

Recommendation:

- Update `amountCollected` to zero before external calls.
- Alternatively, implement OpenZeppelin's `ReentrancyGuard`.

Code Reference:

```
function _refundDonators(uint _id) internal {
    uint256 donationAmount;
    Campaign storage campaign = campaigns[_id];
    campaign.amountCollected = 0; // Move before loop
    for (uint i; i < campaign.donators.length; i++) {
        donationAmount = campaign.donations[i];
        campaign.donations[i] = 0;
        _payTo(campaign.donators[i], donationAmount);
    }
}
```

Finding 2: Logical Flaw in deleteCampaign Allowing Double-Spending

Severity: High
Status: Open

Description: The `deleteCampaign` function does not verify if `campaigns[_id].payedOut` is `true` before calling `_refundDonators`. A campaign owner can call `payOutToCampaignTeam` to withdraw funds and then `deleteCampaign` to trigger additional refunds, double-spending the campaign's funds.

Impact:

- Depletes contract balance, affecting other campaigns' payouts or refunds.
- Potential financial loss for the platform.

Prerequisites:

- Campaign has non-zero `amountCollected`.
- Campaign owner has access to `payOutToCampaignTeam`.

Proof of Concept:

1. Owner creates a campaign (ID 1) and collects 1 ETH from donators.
2. Owner calls `payOutToCampaignTeam(1)`, receiving 0.9 ETH (after 10% tax).
3. Owner calls `deleteCampaign(1)`, triggering `_refundDonators` to refund 1 ETH from the contract's balance.
4. Total received: 1.9 ETH, causing a 0.9 ETH loss to the contract.

Recommendation:

Reset `amountCollected` to zero in `payOutToCampaignTeam` before transferring funds.

Code Reference:

```
function payOutToCampaignTeam(uint256 _id) external privilageEntity(_id)
    if (campaigns[_id].payedOut == true) revert("Funds withdrawed before"
    if (msg.sender != address(owner()) && campaigns[_id].deadline > block
        revert DeadLine({
            campaingDeadline: campaigns[_id].deadline,
            requestTime: block.timestamp
        });
    }
    campaigns[_id].payedOut = true;
    (uint256 raisedAmount, uint256 taxAmount) = _calculateTax(_id);
    campaigns[_id].amountCollected = 0; // Reset amountCollected
    _payTo(campaigns[_id].owner, (raisedAmount - taxAmount));
    _payPlatformFee(taxAmount);
    emit Action(_id, "Funds Withdrawal", msg.sender, block.timestamp);
    return true;
}
```

Finding 3: Campaign ID Overlap Due to Decrementing numberOfCampaigns

Severity: High
Status: Open

Description:

The `deleteCampaign` function decrements `numberOfCampaigns`, causing new campaign IDs to overlap with existing ones, overwriting their data. For example, if `numberOfCampaigns` is 10, a new campaign gets ID 10, and after a deletion(for example with ID 5), it drops to 9, the next campaign reuses ID 10.

Impact:

- Overwrites existing campaign data, including funds, leading to loss of user contributions.
- Compromises contract integrity and user trust.

Prerequisites:

- Multiple campaigns exist in the contract.
- At least one campaign is deleted.

Proof of Concept:

1. Create campaigns until `numberOfCampaigns` reaches 10, assigning ID 10 to campaign C1 with 1 ETH collected.
2. Delete another campaign (e.g., ID 5), reducing `numberOfCampaigns` to 9.
3. Create a new campaign (C2), which reuses ID 10, overwriting C1's data, including its funds and donators.
4. C1's data and funds are lost.

Recommendation:

Remove `numberOfCampaigns -= 1` from `deleteCampaign` to ensure unique IDs.

Code Reference:

```
function deleteCampaign(uint256 _id) external privilageEntity(_id) notInE
    require(campaigns[_id].owner > address(0), "No campaign exist with th
    if (campaigns[_id].amountCollected > 0 && !campaigns[_id].payedOut) {
        _refundDonators(_id);
    }
    delete campaigns[_id];
    emit Action(_id, "Campaign Deleted", msg.sender, block.timestamp);
    // Remove: numberOfCampaigns -= 1;
    return true;
}
```

Finding 4: Incorrect Campaign ID in Action Event of createCampaign

Severity: Low
Status: Open

Description: The `Action` event in `createCampaign` emits `numberOfCampaigns` as the campaign ID, which is incorrect since the actual ID is `numberOfCampaigns - 1`.

Impact:

- Causes confusion in off-chain systems or user interfaces relying on event data.
- No direct financial impact but affects user experience.

Prerequisites: None (occurs in every campaign creation).

Proof of Concept:

1. Call `createCampaign` when `numberOfCampaigns = 10`.
2. The campaign is stored at ID 10, but the `Action` event emits ID 11.
3. Off-chain systems (e.g., dApps) display incorrect campaign IDs, leading to user confusion.

Recommendation: Emit `numberOfCampaigns - 1` in the `Action` event.
Code Reference:

```
function createCampaign(...) external ... returns (uint256) {
    require(block.timestamp < _deadline, "Deadline must be in the future"
    Campaign storage campaign = campaigns[numberOfCampaigns];
    numberOfCampaigns++;
    // ... (other assignments)
    emit Action(
        numberOfCampaigns - 1,
        "Campaign Created",
        msg.sender,
        block.timestamp
    );
    return numberOfCampaigns - 1;
}
```

Finding 5: Incorrect Logic in Donation Limit Check in donateToCampaign

Severity: Medium
Status: Open

Description:

The condition `campaign.amountCollected > campaign.amountCollected.add(amount)` in `donateToCampaign` is always false due to its logical flaw, allowing donations to exceed the campaign's `target`.

Impact:

- Violates the intended logic of limiting donations to the target amount.
- Causes confusion for users and potential issues in off-chain systems expecting capped donations.

Prerequisites: Campaign has a defined `target`.

Proof of Concept:

1. Create a campaign with `target = 1 ETH`.
2. Donate 0.6 ETH, then attempt to donate 0.5 ETH (total: 1.1 ETH).
3. The transaction succeeds despite exceeding `target`, violating the contract's logic.

Recommendation: Use `campaign.amountCollected.add(amount) > campaign.target` to enforce the donation limit.

Code Reference:

```
function donateToCampaign(uint256 _id) external payable notInEmergency {

    // ...

    uint256 amount = msg.value;
    if (campaign.amountCollected.add(amount) > campaign.target)
        revert("Target amount has reached");
    campaign.amountCollected = campaign.amountCollected.add(amount);
    campaign.donators.push(msg.sender);
    campaign.donations.push(amount);
    emit Action(_id, "Donation To The Campaign", msg.sender, block.timest
}
```

Finding 6: Potential Rounding Errors in _calculateTax Due to Low Precision

Severity: Low
Status: Open

Description:

The tax calculation in `_calculateTax` uses a denominator of 100, leading to rounding errors (e.g., 0.9 wei lost for 1,234,567,890,123,456,789 wei).

Impact:

- Minor financial inaccuracies for the platform, potentially accumulating over many campaigns.
- May result in slightly higher payouts to campaign owners.

Prerequisites:
`amountCollected` values not divisible by 100.

Proof of Concept:

1. Create a campaign with `amountCollected = 1,007 wei`.
2. Call `_calculateTax`: `(1,007 * 10) / 100 = 100 wei` (expected: 100.7 wei).
3. Loss of 0.7 wei per campaign, accumulating over multiple campaigns (e.g., 0.000007 ETH for 10,000 campaigns).

Recommendation:
Use a denominator of 10,000 for higher precision.

Finding 7: Bypassing Campaign Halt in updateCampaign When amountCollected is Zero

Severity: Low
Status: Open

Description:

A campaign owner can bypass a halt applied via `haltCampaign` by calling `updateCampaign` to reset `deadline`, but only if `amountCollected` is zero.

Impact:

- Undermines the contract owner's authority to halt campaigns.
- Limited to unfunded campaigns, reducing financial risk.

Prerequisites:

- Campaign is halted by the contract owner.
- Campaign has `amountCollected = 0`.

Proof of Concept:

1. Contract owner calls `haltCampaign(1)`, setting `deadline = block.timestamp`.

2. Campaign owner calls `updateCampaign(1)` to reset `deadline` to a future timestamp, as `amountCollected = 0`.
3. Campaign becomes active again, bypassing the halt.

Recommendation:
Add an `isHalted` flag to prevent updates on halted campaigns.

Code Reference:

```
// Add to Campaign struct: bool isHalted;
function haltCampaign(uint256 _id) external onlyOwner {
    campaigns[_id].deadline = block.timestamp;
    campaigns[_id].isHalted = true;
    emit Action(_id, "Campaign halted", msg.sender, block.timestamp);
}

function updateCampaign(...) external ... {
    require(!campaign.isHalted, "Campaign is halted and cannot be updatec
    // ... (rest of function)
}
```

Finding 8: Inclusion of Deleted and Paid-Out Campaigns in getCampaigns Output

Severity: Informational
Status: Open

Description: The `getCampaigns` function returns all campaigns, including deleted (with default struct values) and paid-out campaigns, without filtering.

Impact:

- Causes confusion for users or dApps expecting only active campaigns.
- Increases gas costs and data clutter in off-chain systems.

Prerequisites: Deleted or paid-out campaigns exist.

Recommendation:

Filter campaigns by `owner != address(0)` and `!payedOut`. Consider pagination for scalability.

Finding 9: Inability to Refund All Campaigns in Emergency Mode Due to Off-by-One Error

Severity: Medium

Status: Open

Description: In emergency mode (`emergencyMode = true`), the `withdrawFunds` function calls `_refundDonators(_startId, _endId)` , which iterates from `_startId` to `_endId` (excluding `_endId`). This off-by-one error prevents the campaign at `_endId` from being refunded. Additionally, the condition `require(_idFrom < _idTo)` and `campaigns[_idTo].owner > address(0)` means that if only one campaign exists (e.g., ID 0), it cannot be refunded, as `_idTo` must be greater than `_idFrom` . This issue generalizes to cases where `_idFrom = 0` , leaving one valid campaign unrefunded.

Impact:

- Prevents complete refunds in emergency mode, violating the intended emergency logic.
- Reduces user trust, especially in single-campaign scenarios or when `_idFrom = 0` .

Prerequisites:

- `emergencyMode = true` .
- Single campaign (ID 0) or `_idFrom = 0` with multiple campaigns.

Proof of Concept:

1. Enable `emergencyMode` .
2. Create a single campaign (ID 0) with 1 ETH donated.
3. Call `withdrawFunds(0, 1)` ; transaction reverts due to `require(_idFrom < _idTo)` and `campaigns[1].owner > address(0)` failing (no campaign at ID 1).
4. Alternatively, with campaigns at IDs 0 and 1, call `withdrawFunds(0, 1)` ; only ID 0 is refunded, leaving ID 1 unrefunded.
5. No combination allows refunding all campaigns when `_idFrom = 0` .

Recommendation:

- Modify `_refundDonators` to include `_endId` in the loop (`i <= _endId`).
- Remove `require(_idFrom < _idTo)` to allow single-campaign refunds.
  Code Reference:

```
function _refundDonators(uint256 _idFrom, uint256 _idTo) internal {
    require(campaigns[_idFrom].owner > address(0), "No campaign exist wit
    // Remove: require(_idFrom < _idTo, "Invalid range");
    for (uint256 i = _idFrom; i <= _idTo; i++) { // Include _idTo
        if (campaigns[i].owner == address(0)) continue;
        uint256 donationAmount;
```

```
            Campaign storage campaign = campaigns[i];
            campaign.amountCollected = 0;
            for (uint j; j < campaign.donators.length; j++) {
                donationAmount = campaign.donations[j];
                campaign.donations[j] = 0;
                _payTo(campaign.donators[j], donationAmount);
            }
        }
    }
```

Conclusion

The `CrowdFunding.sol` contract contains critical vulnerabilities in fund management, campaign ID allocation, and emergency refund logic, alongside logical and informational issues. The High-severity issues (Findings 1-3) pose significant risks to financial integrity and must be prioritized. The Medium-severity issues (Findings 5, 9) and Low-severity issues (Findings 4, 6, 7) should be addressed to enhance reliability and user experience. The Informational issue (Finding 8) can improve off-chain integration.

I am available to discuss these findings or assist with implementing fixes. Please contact me at [Your Email] or GitHub ([Your GitHub Username]).

Acknowledgments

Thank you to the CrowdFunding team for developing this open-source project, enabling community-driven security improvements.