# PartyDAO contest
# Findings & Analysis Report

2022-11-17

## Table of contents

## Overview

## About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the PartyDAO smart contract system written in Solidity. The audit contest took place between September 12—September 19 2022.

## Wardens

119 Wardens contributed reports to the PartyDAO contest:

1. Lambda
2. Trust
3. 0x52
4. csanuragjain
5. cccz
6. 8olidity
7. bin2chen
8. hyh
9. ladboy233
10. CertoraInc (egjlmn1, OriDabush, ItayG, shakedwinder and RoiEvenHaim)
11. smiling_heretic
12. V_B (Barichek and vlad_bochok)
13. rvierdiiev
14. 0xA5DF
15. Ch_301
16. Jeiwan
17. 0xSmartContract
18. tonisives

19. rbserver
20. Chom
21. pfapostol
22. brgltd
23. c3phas
24. 0xNazgul
25. m_Rassska
26. JC
27. Deivitto
28. CodingNameKiki
29. ayeslick
30. MiloTruck
31. ReyAdmirado
32. gogo
33. 0x1f8b
34. CRYP70
35. asutorufos
36. bulej93
37. RaymondFam
38. erictee
39. leosathya
40. __141345__
41. Aymen0909
42. fatherOfBlocks
43. lukris02
44. Funen
45. malinariy
46. 0x5rings
47. PaludoX0

48. ch0bu
49. ChristianKuri
50. slowmoses
51. delfin454000
52. djxploit
53. martin
54. Tomo
55. Olivierdem
56. 0x4non
57. tnevler
58. B2
59. StevenL
60. d3e4
61. cryptphi
62. R2
63. 0xbepresent
64. hansfriese
65. pedr02b2
66. wagmi
67. KIntern_NA (TrungOre and duc)
68. The*GUILD ([David*](https://twitter.com/davidpius10), Ephraim, LeoGold, and greatsamist)
69. Anth3m
70. 0xDanielC
71. JansenC
72. MasterCookie
73. indijanc
74. Rolezn
75. 0xkatana
76. JAGADESH

77. Sm4rty
78. ajtra
79. peiw
80. SnowMan
81. prasantgupta52
82. Rohan16
83. Bnke0x0
84. gianganhnguyen
85. karanctf
86. Tomio
87. sryysryy
88. ignacio
89. Metatron
90. Waze
91. robee
92. Saintcode_
93. jag
94. got_targ
95. Amithuddar
96. Fitraldys
97. LeoS
98. natzuu
99. simon135
100. Diraco
101. francoHacker
102. Noah3o6
103. Ocean_Sky
104. dharma09
105. IgnacioB

106. peanuts
107. 0x85102
108. aysha
109. Matin
110. pashov

This contest was judged by HardlyDifficult.

Final report assembled by itsmetechjay.

## Summary

The C4 analysis yielded an aggregated total of 19 unique vulnerabilities. Of these vulnerabilities, 7 received a risk rating in the category of HIGH severity and 12 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 67 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 82 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

## Scope

The code under review can be found within the C4 PartyDAO contest repository, and is composed of 46 smart contracts written in the Solidity programming language and includes 3,995 lines of Solidity code. For the purposes of the audit contest, the code was forked into a new repo from its main location in the PartyDAO party-protocol repo at this commit hash: 1eb4909a2f568d910e70e2db1b487a0236fbe10b.

## Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

## High Risk Findings (7)

### [H-01] PartyGovernance: Can vote multiple times by transferring NFT in same block as proposal

*Submitted by Lambda, also found by Trust*

`PartyGovernanceNFT` uses the voting power at the time of proposal when calling `accept`. The problem with that is that a user can vote, transfer the NFT (and the voting power) to a different wallet, and then vote from this second wallet again during the same block that the proposal was created. This can also be repeated multiple times to get an arbitrarily high voting power and pass every proposal unanimously.

The consequences of this are very severe. Any user (no matter how small his voting power is) can propose and pass arbitrary proposals animously and therefore steal all assets (including the precious tokens) out of the party.

#### Proof Of Concept

This diff shows how a user with a voting power of 50/100 gets a voting power of 100/100 by transferring the NFT to a second wallet that he owns and voting from that one:

```
--- a/sol-tests/party/PartyGovernanceUnit.t.sol
+++ b/sol-tests/party/PartyGovernanceUnit.t.sol
```

```
@@ -762,6 +762,7 @@ contract PartyGovernanceUnitTest is Test, Te
         TestablePartyGovernance gov =
             _createGovernance(100e18, preciousTokens, preciousT
         address undelegatedVoter = _randomAddress();
+        address recipient = _randomAddress();
         // undelegatedVoter has 50/100 intrinsic VP (delegated 
         gov.rawAdjustVotingPower(undelegatedVoter, 50e18, addre

@@ -772,38 +773,13 @@ contract PartyGovernanceUnitTest is Test, 
         // Undelegated voter submits proposal.
         vm.prank(undelegatedVoter);
         assertEq(gov.propose(proposal, 0), proposalId);
-
-        // Try to execute proposal (fail).
-        vm.expectRevert(abi.encodeWithSelector(
-            PartyGovernance.BadProposalStatusError.selector,
-            PartyGovernance.ProposalStatus.Voting
-        ));
-        vm.prank(undelegatedVoter);
-        gov.execute(
-            proposalId,
-            proposal,
-            preciousTokens,
-            preciousTokenIds,
-            "",
-            ""
-        );
-
-        // Skip past execution delay.
-        skip(defaultGovernanceOpts.executionDelay);
-        // Try again (fail).
-        vm.expectRevert(abi.encodeWithSelector(
-            PartyGovernance.BadProposalStatusError.selector,
-            PartyGovernance.ProposalStatus.Voting
-        ));
-        vm.prank(undelegatedVoter);
-        gov.execute(
-            proposalId,
-            proposal,
-            preciousTokens,
-            preciousTokenIds,
-            "",
-            ""
-        );
+        (, PartyGovernance.ProposalStateValues memory valuesPre
+        assertEq(valuesPrev.votes, 50e18);
+           gov.transferVotingPower(undelegatedVoter, recipient, 50
+           vm.prank(recipient);
+           gov.accept(proposalId, 0);
+           (, PartyGovernance.ProposalStateValues memory valuesAft
+           assertEq(valuesAfter.votes, 100e18);
        }
```

## Recommended Mitigation Steps

You should query the voting power at `values.proposedTime - 1`. This value is already finalized when the proposal is created and therefore cannot be manipulated by repeatedly transferring the voting power to different wallets.

**merklejerk (PartyDAO) confirmed and commented:**

> This is our favorite find and want to call it out specifically. We would consider this critical.

> We will implement the suggested fix in this PR and use `proposedTime - 1` for voting power calculations.

**HardlyDifficult (judge) commented:**

> Agree with High risk - any user with a non-zero voting power can pass a proposal & steal assets.

**0xble (PartyDAO) resolved:**

> Resolved: https://github.com/PartyDAO/partybidV2/pull/130

## [H-02] Possibility to burn all ETH in Crowdfund under some circumstances

*Submitted by Lambda, also found by 8olidity*

If `opts.initialContributor` is set to `address(0)` (and `opts.initialDelegate` is not), there are two problems: 1.) If the crowdfund succeeds, the initial balance will be lost. It is still accredited to `address(0)`, but it is not retrievable. 2.) If the crowdfund does not succeed, anyone can completely drain the contract by repeatedly calling `burn` with `address(0)`. This will always succeed because `CrowdfundNFT._burn` can be called multiple times for `address(0)`. Every call will cause the initial balance to be burned (transferred to `address(0)`).

Issue 1 is somewhat problematic, but issue 2 is very problematic, because all funds of a crowdfund are burned and an attacker can specifically set up such a deployment (and the user would not notice anything special, after all these are parameters that the protocol accepts).

## Proof Of Concept

This diff illustrates scenario 2, i.e. where a malicious deployer burns all contributions (1 ETH) of `contributor`. He loses 0.25ETH for the attack, but this could be reduced significantly (with more `burn(payable(address(0)))` calls:

```
--- a/sol-tests/crowdfund/BuyCrowdfund.t.sol
+++ b/sol-tests/crowdfund/BuyCrowdfund.t.sol
@@ -36,9 +36,9 @@ contract BuyCrowdfundTest is Test, TestUtils {
     string defaultSymbol = 'PBID';
     uint40 defaultDuration = 60 * 60;
     uint96 defaultMaxPrice = 10e18;
-    address payable defaultSplitRecipient = payable(0);
+    address payable defaultSplitRecipient = payable(address(thi
     uint16 defaultSplitBps = 0.1e4;
-    address defaultInitialDelegate;
+    address defaultInitialDelegate = address(this);
     IGateKeeper defaultGateKeeper;
     bytes12 defaultGateKeeperId;
     Crowdfund.FixedGovernanceOpts defaultGovernanceOpts;
@@ -78,7 +78,7 @@ contract BuyCrowdfundTest is Test, TestUtils {
                     maximumPrice: defaultMaxPrice,
                     splitRecipient: defaultSplitRecipient,
                     splitBps: defaultSplitBps,
-                    initialContributor: address(this),
+                    initialContributor: address(0),
                     initialDelegate: defaultInitialDelegate,
                     gateKeeper: defaultGateKeeper,
                     gateKeeperId: defaultGateKeeperId,
@@ -111,40 +111,26 @@ contract BuyCrowdfundTest is Test, TestUti
     function testHappyPath() public {
         uint256 tokenId = erc721Vault.mint();
```

```
          // Create a BuyCrowdfund instance.
-         BuyCrowdfund pb = _createCrowdfund(tokenId, 0);
+         BuyCrowdfund pb = _createCrowdfund(tokenId, 0.25e18);
          // Contribute and delegate.
          address payable contributor = _randomAddress();
          address delegate = _randomAddress();
          vm.deal(contributor, 1e18);
          vm.prank(contributor);
          pb.contribute{ value: contributor.balance }(delegate, "
-         // Buy the token.
-         vm.expectEmit(false, false, false, true);
-         emit MockPartyFactoryCreateParty(
-             address(pb),
-             address(pb),
-             _createExpectedPartyOptions(0.5e18),
-             _toERC721Array(erc721Vault.token()),
-             _toUint256Array(tokenId)
-         );
-         Party party_ = pb.buy(
-             payable(address(erc721Vault)),
-             0.5e18,
-             abi.encodeCall(erc721Vault.claim, (tokenId)),
-             defaultGovernanceOpts
-         );
-         assertEq(address(party), address(party_));
-         // Burn contributor's NFT, mock minting governance toke
-         // unused contribution.
-         vm.expectEmit(false, false, false, true);
-         emit MockMint(
-             address(pb),
-             contributor,
-             0.5e18,
-             delegate
-         );
-         pb.burn(contributor);
-         assertEq(contributor.balance, 0.5e18);
+         vm.warp(block.timestamp + defaultDuration + 1);
+         // The auction was not won, we can now burn all ETH fro
+         assertEq(address(pb).balance, 1.25e18);
+         pb.burn(payable(address(0)));
+         assertEq(address(pb).balance, 1e18);
+         pb.burn(payable(address(0)));
+         assertEq(address(pb).balance, 0.75e18);
+         pb.burn(payable(address(0)));
+         assertEq(address(pb).balance, 0.5e18);
+         pb.burn(payable(address(0)));
+           assertEq(address(pb).balance, 0.25e18);
+           pb.burn(payable(address(0)));
+           assertEq(address(pb).balance, 0);
```

### Recommended Mitigation Steps

Do not allow an initial contribution when `opts.initialContributor` is not set.

[merklejerk (PartyDAO) confirmed and commented](#):

> Excellent catch. We will implement the fix from [#238](#) and prevent minting to `address(0)`.

[HardlyDifficult (judge) commented](#):

> Agree with High risk - a crowdfund's initial configuration could lead to the loss of user funds.

[0xble (PartyDAO) resolved](#):

> Resolved: [https://github.com/PartyDAO/partybidV2/pull/127](https://github.com/PartyDAO/partybidV2/pull/127)

## [H-03] A majority attack can easily bypass Zora auction stage in OpenseaProposal and steal the NFT from the party.

*Submitted by Trust*

The PartyGovernance system has many defenses in place to protect against a majority holder stealing the NFT. One of the main protections is that before listing the NFT on Opensea for a proposal-supplied price, it must first try to be auctioned off on Zora. To move from Zora stage to Opensea stage, `\_settleZoraAuction()` is called when executing ListedOnZora step in ListOnOpenseaProposal.sol. If the function returns false, the next step is executed which lists the item on Opensea. It is assumed that if majority attack proposal reaches this stage, it can steal the NFT for free, because it can list the item for negligible price and immediately purchase it from a contract that executes the Opensea proposal.

Indeed, attacker can always make `settleZoraAuction()` return false. Looking at the code:

```
    try ZORA.endAuction(auctionId) {
                // Check whether auction cancelled due to a failed t
                // settlement by seeing if we now possess the NFT.
                if (token.safeOwnerOf(tokenId) == address(this)) {
                    emit ZoraAuctionFailed(auctionId);
                    return false;
                }
            } catch (bytes memory errData) {
```

As the comment already hints, an auction can be cancelled if the NFT transfer to the bidder fails. This is the relevant AuctionHouse code (endAuction):

```
    {
                // transfer the token to the winner and pay out the 
                try IERC721(auctions[auctionId].tokenContract).safeT
                    _handleOutgoingBid(auctions[auctionId].bidder, a
                    _cancelAuction(auctionId);
                    return;
     }
```

As most NFTs inherit from OpenZeppelin's ERC721.sol code, safeTransferFrom will run:

```
        function _safeTransfer(
            address from,
            address to,
            uint256 tokenId,
            bytes memory data
        ) internal virtual {
            _transfer(from, to, tokenId);
            require(_checkOnERC721Received(from, to, tokenId, data),
        }
```

So, attacker can bid a very high amount on the NFT to ensure it is the winning bid. When AuctionHouse tries to send the NFT to attacker, the safeTransferFrom will fail because attack will not implement an ERC721Receiver. This will force the AuctionHouse to return the bid amount to the bidder and cancel the auction.

Importantly, it will lead to a graceful return from `endAuction()`, which will make `settleZoraAuction()` return false and progress to the OpenSea stage.

## Impact

A majority attack can easily bypass Zora auction stage and steal the NFT from the party.

## Proof of Concept

1. Pass a ListOnOpenseaProposal with a tiny list price and execute it
2. Create an attacker contract which bids on the NFT an overpriced amount, but does not implement ERC721Receiver. Call its bid() function
3. Wait for the auction to end ( timeout after the bid() call)
4. Create a contract with a function which calls execute() on the proposal and immediately buys the item on Seaport. Call the attack function.

## Recommended Mitigation Steps

`\_settleZoraAuction` is called from both ListOnZoraProposal and ListOnOpenseaProposal. If the auction was cancelled due to a failed transfer, as is described in the comment, we would like to handle it differently for each proposal type. For ListOnZoraProposal, it should indeed return false, in order to finish executing the proposal and not to hang the engine. For ListOnOpenseaProposal, the desired behavior is to *revert* in the case of a failed transfer. This is because the next stage is risky and defense against the mentioned attack is required. Therefore, pass a revertOnFail flag to `\_settleZoraAuction`, which will be used like so:

```
    // Check whether auction cancelled due to a failed transfer durin
    // settlement by seeing if we now possess the NFT.
    if (token.safeOwnerOf(tokenId) == address(this)) {
            if (revertOnFail) {
                    revert("Zora auction failed because of transfer 
            }
               emit ZoraAuctionFailed(auctionId);
               return false;
    }
```

merklejerk (PartyDAO) confirmed and commented:

> Great find. We will modify `_settleZoraAuction()` to return some auction status to be communicated up to the Opensea proposal.

HardlyDifficult (judge) commented:

> TIL. While digging into this I noticed that Zora changed this logic in their V3 implementation, avoiding this scenario - but there may be reasons to prefer the auction house contract.

> Agree with High risk - the auction safeguard can be bypassed, allowing a majority owner to steal from the rest of the party.

0xble (PartyDAO) resolved:

> Resolved: https://github.com/PartyDAO/partybidV2/pull/137

## [H-04] TokenDistributor: ERC777 tokensToSend hook can be exploited to drain contract

*Submitted by Lambda*

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/distribution/TokenDistributor.sol#L131

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/distribution/TokenDistributor.sol#L386

### Impact

`TokenDistributor.createERC20Distribution` can be used to create token distributions for ERC777 tokens (which are backwards-compatible with ERC20). However, this introduces a reentrancy vulnerability which allows a party to get the tokens of another party. The problem is the `tokensToSend` hook which is executed BEFORE balance updates happens (see https://eips.ethereum.org/EIPS/eip-777). When this hook is executed, `token.balanceOf(address(this))` therefore still returns the old value, but `_storedBalances[balanceID]` was already decreased.

## Proof Of Concept

Party A and Party B have a balance of 1,000,000 tokens (of some arbitrary ERC777 token) in the distributor. Let's say for the sake of simplicity that both parties only have one user (user A in party A, user B in party B). User A (or rather his smart contract) performs the following attack:

- He calls `claim`, which transfers 1,000,000 tokens to his contract address. In `_transfer`, `_storedBalances[balanceId]` is decreased by 1,000,000 and therefore now has a value of 1,000,000.
- In the `tokensToSend` hook, he initiates another distribution for his party A by calling `PartyGovernance.distribute` which calls `TokenDistributor.createERC20Distribution` (we assume for the sake of simplicity that the party does not have more of these tokens, so the call transfers 0 tokens to the distributor). `TokenDistributor.createERC20Distribution` passes `token.balanceOf(address(this))` to `_createDistribution`. Note that this is still 2,000,000 because we are in the `tokensToSend` hook.
- The supply of this distribution is calculated as `(args.currentTokenBalance - _storedBalances[balanceId]) = 2,000,000 - 1,000,000 = 1,000,000`.
- When the `tokensToSend` hook is exited (and the first transfer has finished), he can retrieve the tokens of the second distribution (that was created in the hook) to steal the 1,000,000 tokens of party B.

## Recommended Mitigation Steps

Do not allow reentrancy in these functions.

[merklejerk (PartyDAO) confirmed and commented]():

> Very few legitimate ERC777s so we think the probability of this happening to a party is somewhat low. Also, it only impacts distributions for that token. However, we will be implementing a reentrancy guard to fix it.

[HardlyDifficult (judge) commented]():

> Agree that it does not seem very probable - but if 777 assets are distributed, it does appear to be a way of stealing from other users in the party and therefore High risk.

0xble (PartyDAO) resolved:

> Resolved: https://github.com/PartyDAO/partybidV2/pull/132

## [H-05] ArbitraryCallsProposal.sol and ListOnOpenseaProposal.sol safeguards can be bypassed by cancelling in-progress proposal allowing the majority to steal NFT

*Submitted by 0x52*

Note: PartyDAO acknowledges that "canceling an InProgress proposal (mid-step) can leave the governance party in a vulnerable or undesirable state because there is no cleanup logic run during a cancel" in the "Known Issues / Topics" section of the contest readme. I still believe that this vulnerability needs to be mitigated as it can directly lead to loss of user funds.

### Impact

Majority vote can abuse cancel functionality to steal an NFT owned by the party.

### Proof of Concept

ArbitraryCallsProposal.sol implements the following safeguards for arbitrary proposals that are not unanimous:

1. Prevents the ownership of any NFT changing during the call. It does this by checking the the ownership of all NFTs before and after the call.
2. Prevents calls that would change the approval status of any NFT. This is done by disallowing the "approve" and "setApprovalForAll" function selectors.

Additionally ListOnOpenseaProposal.sol implements the following safeguards:

1. NFTs are first listed for auction on Zora so that if they are listed for a very low price then the auction will keep them from being purchased at such a low price.
2. At the end of the auction the approval is revoked when `\_cleanUpListing` is called.

These safeguards are ultimately ineffective though. The majority could use the following steps to steal the NFT:

1. Create ListOnOpenseaProposal with high sell price and short cancel delay
2. Vote to approve proposal with majority vote
3. Execute first step of proposal, listing NFT on Zora auction for high price
4. Wait for Zora auction to end since the auction price is so high that no one will buy it
5. Execute next step, listing the NFT on Opensea. During this step the contract grants approval of the NFT to the Opensea contract
6. Wait for cancelDelay to expire
7. Call PartyGovernance.sol#cancel. This will immediately terminate the Opensea bypassing `\_cleanUpListing` and keeping the approval to the Opensea contract.
8. Create ArbitraryCallsProposal.sol that lists the NFT on Opensea for virtually nothing. Since only approval selectors have been blacklisted and the NFT does not change ownership, the proposal does not need to be unanimous to execute.
9. Approve proposal and execute.
10. Buy NFT.

## Recommended Mitigation Steps

When a proposal is canceled, it should call a proposal specific function that makes sure everything is cleaned up. NFTs delisted, approvals revoked, etc.

merklejerk (PartyDAO) confirmed and commented:

> We will block calls to `opensea.validate()` in Arbitrary call proposals.

HardlyDifficult (judge) commented:

> Agree with High risk - in this scenario a majority owner could steal the asset from others in the party.

0xble (PartyDAO) resolved:

> Resolved: [https://github.com/PartyDAO/partybidV2/pull/139](https://github.com/PartyDAO/partybidV2/pull/139)

## [H-06] A majority attack can steal precious NFT from the party by crafting and chaining two proposals

*Submitted by Trust, also found by ladboy233 and Lambda*

[https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/proposals/ProposalExecutionEngine.sol#L116](https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/proposals/ProposalExecutionEngine.sol#L116)

[https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/proposals/FractionalizeProposal.sol#L54-L62](https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/proposals/FractionalizeProposal.sol#L54-L62)

### Description

The PartyGovernance system has many defenses in place to protect against a majority holder stealing the NFT. Majority cannot exfiltrate the ETH gained from selling precious NFT via any proposal, and it's impossible to sell NFT for any asset except ETH. If the party were to be compensated via an ERC20 token, majority could pass an ArbitraryCallsProposal to transfer these tokens to an attacker wallet. Unfortunately, FractionalizeProposal is vulnerable to this attack. Attackers could pass two proposals and wait for them to be ready for execution. Firstly, a FractionalizeProposal to fractionalize the NFT and mint totalVotingPower amount of ERC20 tokens of the created vault. Secondly, an ArbitraryCallsProposal to transfer the entire ERC20 token supply to an attacker address. At this point, attacker can call `vault.redeem()` to burn the outstanding token supply and receive the NFT back.

### Impact

A 51% majority could steal the precious NFT from the party and leave it empty.

### Proof of Concept

The only non-trivial component of this attack is that the created vault, whose tokens we wish to transfer out, has an undetermined address until `VAULT_FACTORY.mint()` is called, which creates it. The opcode which creates the vault contract is CREATE, which calculates the address with `keccak256(VAULT_FACTORY, nonce)`. Nonce will keep changing while new, unrelated NFTs are fractionalized. The attack needs to prepare both FractionalizedProposal and ArbitraryCallsProposal ahead of time, so that they could be chained immediately, meaning there would be no time for other members to call `distribute()` on the party, which would store the fractionalized tokens safely in the distributor. In order to solve this chicken and the egg problem, we will use a technique taken from traditional low-level exploitation called heap feng shui.

Firstly, calculate off-chain, the rate new NFTs are fractionalized, and multiple by a safety factor (like 1.2X), and multiply again by the proposal execution delay. This number, added to the current `VAULT_FACTORY` nonce, will be our `target_nonce`. Calculate `target_vault = keccak256(VAULT_FACTORY, target_nonce)`, `before_target_vault = keccak256(VAULT_FACTORY, target_nonce-1)`

Firstly, we will create a contract which has an attack function that:

1. Loop while before*target*vault != created*vault: • Mint new dummy attacker*NFT • created*vault = VAULT*FACTORY.mint(attacker_NFT...)
2. `Call execute()` on the FractionalizedProposal // We will feed the execute() parameters to the contract in a separate contract setter. Note that this is guaranteed to create target_vault on the correct address.
3. `Call execute()` on the ArbitraryCallsProposal

Then, we propose the two proposals:

1. Propose a FractionalizedProposal, with any list price and the precious NFT as parameter
2. Propose an ArbitraryCallsProposal, with target = target_vault, data = transfer(ATTACKER, totalVotingPower)

Then, we set the `execute()` parameters passed in step 2 and 3 of the attack contract using the proposalID allocated for them.

Then, we wait for execution delay to finish.

Finally, run the `attack()` function prepared earlier. This will increment the `VAULT_FACTORY` nonce until it is the one we count on during the ArbitraryCallsProposal. Pass enough gas to be able to burn enough nonces.

At this point, attacker has all the vault tokens, so he may call vault.redeem() and receive the precious NFT.

## Recommended Mitigation Steps

1. Enforce a minimum cooldown between proposals. This will mitigate additional weaknesses of the proposal structure. Here, this will give users the opportunity to call `distribute()` to put the vault tokens safe in distributor.
2. A specific fix here would be to call `distribute()` at the end of FractionalizeProposal so that there is no window to steal the funds.

**merklejerk (PartyDAO) confirmed and commented:**

> Will fix by creating an automatic distribution at the end of a successful fractionalize proposal.

**HardlyDifficult (judge) commented:**

> Agree with High risk - this scenario allows a majority owner to steal from others in the party.

**0xble (PartyDAO) resolved:**

> Resolved: https://github.com/PartyDAO/partybidV2/pull/131

## [H-07] Attacker can DOS private party by donating ETH then calling buy

*Submitted by 0x52*

Party is DOS'd and may potentially lose access to NFT.

## Proof of Concept

Crowdfund.sol#L280-L298

```
party = party_ = partyFactory
    .createParty(
        address(this),
        Party.PartyOptions({
            name: name,
            symbol: symbol,
            governance: PartyGovernance.GovernanceOpts({
                hosts: governanceOpts.hosts,
                voteDuration: governanceOpts.voteDuration,
                executionDelay: governanceOpts.executionDelay,
                passThresholdBps: governanceOpts.passThresholdBp
                totalVotingPower: _getFinalPrice().safeCastUint2
                feeBps: governanceOpts.feeBps,
                feeRecipient: governanceOpts.feeRecipient
            })
        }),
        preciousTokens,
        preciousTokenIds
    );
```

[BuyCrowdfundBase.sol#L166-L173]

```
    function _getFinalPrice()
        internal
        override
        view
        returns (uint256)
    {
        return settledPrice;
    }
```

When BuyCrowdFund.sol successfully completes a buy, totalVotingPower is set to `\_getFinalPrice` which in the case of BuyCrowdFundBase.sol returns the price at which the NFT was purchased. totalVotingPower is used by the governance contract to determine the number of votes needed for a proposal to pass. If there are not enough claimable votes to meet that threshold then the party is softlocked because it can't pass any proposals. An attacker could exploit this to DOS even a private party with the following steps:

1. Wait for party to be filled to just under quorum threshold
2. Donate ETH to the crowdfund contract

3. Call BuyCrowdFund.sol#buy. Since it is unpermissioned even for parties with a gatekeeper, the call won't revert

Since the voting power for the final amount of ETH cannot be claimed, the party is now softlocked. If emergencyExecuteDisabled is true then the party will be permanantly locked and the NFT would effectively be burned. If emergencyExecuteDisabled is false then users would have to wait for PartyDAO to reclaim the NFT.

## Recommended Mitigation Steps

Permission to call BuyCrowdFund.sol#buy should be gated if there is a gatekeeper.

**merklejerk (PartyDAO) confirmed and commented:**

> Theoretically possible but there doesn't seem to be much upside for the attacker. We do think it's unusual that buy()/bid() can be called by a non-member for a private/gatekept party, so we will add gatekeeping logic there to fix this. We will also cap the callValue (and therefore final price) to `totalContributions`.

**0xble (PartyDAO) resolved:**

> Resolved: https://github.com/PartyDAO/partybidV2/pull/133

**HardlyDifficult (judge) increased severity to High and commented:**

> Although it's without upside, it is a path for the attacker to potentially lock the NFT. Since it can cause a loss of asset for users, this seems to be a High risk issue.

> Let me know if I misunderstood.

**merklejerk (PartyDAO) commented:**

> ~Don't consider it high because there is a much more straightforward way to softlock a party: contribute normally and don't ever participate in governance.~ Oh nvm, this is the private party one. Yeah I'm fine with high.

## Medium Risk Findings (12)

## [M-01] Attacker can list an NFT they own and inflate to zero all users' contributions, keeping the NFT and all the money

*Submitted by Trust, also found by smiling*\heretic_

Crowdfunds split the voting power and distribution of profits rights according to the percentage used to purchase the NFT. When an attacker lists his own NFT for sale and contributes to it, any sum he contributes will return to him once the sell is executed. This behavior is fine so long as the sell price is fair, as other contributors will receive a fair portion of voting power and equity. Therefore, when a maximum price is defined, it is not considered a vulnerability that an attacker can contribute `maximumPrice - totalContributions` and take a potentially large stake of the Crowdfund, as user's have contributed knowing the potential maximum price.

However, when maximum price is zero, which is allowed in BuyCrowdfund and CollectionBuyCrowdfund, the lister of the NFT can always steal the entirety of the fund and keep the NFT. Attacker can send a massive contribution, buy the NFT and pass a unanimous proposal to approve the NFT to his wallet. Attacker can use a flash loan to finance the initial contribution, which is easily paid back from the NFT lister wallet.

It is important to note that there is no way for the system to know if the Crowdfund creator and the NFT owner are the same entity, and therefore it is critical for the platform to defend users against this scenario.

### Impact

Any crowdfund with zero maximumPrice configured is subject to NFT lister rug pull and complete takeover of the contribution funds.

### Proof of Concept

1. Suppose totalContributions=X. Attacker will prepare flashloan callback which does:

    - `contribute()` with the borrowed funds
    - call BuyCrowdfund's `buy()` with a contract that will purchase the NFT using the large sum

- call NFT lister contract's sendProfit() function which will send attacker the NFT sell amount
- pay back the flash loan

2. Attacker will call flashloan(10,000X) to buy his own NFT and take unanimous holding of created party
3. Attacker will create ArbitraryCallsProposal with a single call to token.approve(ATTACKER, token_id). It is immediately ready for execution because attacker has > 99.99% of votes. Precious approve() is allowed in unanimous proposals.
4. Attacker calls NFT's `transferFrom()` to take back control of the NFT, leaving with the Crowdfund's contribution and the listed NFT.

## Recommended Mitigation Steps

Disable the option to have unlimited maximumPrice for BuyCrowdfund and CollectionBuyCrowdfund contracts. AuctionCrowdfund is already safe by not allowing a zero maximumBid.

merklejerk (PartyDAO) commented:

> Duplicate of #17

HardlyDifficult (judge) decreased severity to QA and commented:

> See dupe for context. Merging with #198.

Trust (warden) commented:

> TBH I don't see enough resemblance to #17 for it to be dupped. This submission discusses specifically maximumPrice = 0 which allows attacker to always claim the entire contribution pool to themselves. Would appreciate a second look. @HardlyDifficult disclaimer - this is my submission.

HardlyDifficult (judge) increased severity to Medium and commented:

> Yes that is fair. This seems to fall under the known issue "It is possible that someone could manipulate parties by contributing ETH and then buying their NFT that they own." - although an extreme example and a unique take on that scenario. However

> given the apparent ease and impact of this attack it seems we can bump this to Medium risk - would be high if not for the general known issue here.

smiling_heretic (warden) commented:

> The same is true for 206 (disclaimer: it's my submission) so I think it should be marked as duplicate of this finding and also upgraded to medium.

HardlyDifficult (judge) commented:

> Agree, thanks @SmilingHeretic.

## [M-02] Previously nominated delegate can reset the delegation

*Submitted by hyh*

`burn()` allows for previously recorded delegate to set himself to be contributor's delegate even if another one was already chosen.

This can be quite material as owner choice for the whole voting power is being reset this way to favor the old delegate.

### Proof of Concept

`\_burn()` can be invoked by anyone on the behalf of any `contributor`:

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/Crowdfund.sol#L167-L171

```
    function burn(address payable contributor)
        public
    {
        return _burn(contributor, getCrowdfundLifecycle(), party
    }
```

It mints the governance NFT for the contributor whenever he has voting power:

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/Crowdfund.sol#L471-L485

```
        if (votingPower > 0) {
            // Get the address to delegate voting power to. If n
            address delegate = delegationsByContributor[contribu
            if (delegate == address(0)) {
                // Delegate can be unset for the split recipient
                // contribute. Self-delegate if this occurs.
                delegate = contributor;
            }
            // Mint governance NFT for the contributor.
            party_.mint(
                contributor,
                votingPower,
                delegate
            );
        }
```

Now `mint()` calls `\_adjustVotingPower()` with a new delegate, redirecting all the intristic power, not just one for that id, ignoring the delegation the `owner` might already have:

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernanceNFT.sol#L120-L133

```
    function mint(
        address owner,
        uint256 votingPower,
        address delegate
    )
        external
        onlyMinter
        onlyDelegateCall
    {
        uint256 tokenId = ++tokenCount;
        votingPowerByTokenId[tokenId] = votingPower;
        _adjustVotingPower(owner, votingPower.safeCastUint256ToI
        _mint(owner, tokenId);
    }
```

I.e. Bob the contributor can take part in the crowdfunding with `contribute()` with small `0.01 ETH` stake, stating Mike as the delegate of his choice with `contribute(Mike, ...)`:

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/Crowdfund.sol#L189-L208

```
    /// @param delegate The address to delegate to for the gover
    /// @param gateData Data to pass to the gatekeeper to prove 
    function contribute(address delegate, bytes memory gateData)
        public
        payable
    {
        _contribute(
            msg.sender,
            msg.value.safeCastUint256ToUint96(),
            delegate,
            // We cannot use `address(this).balance - msg.value`
            // total contributions in case someone forces (suici
            // contract. This wouldn't be such a big deal for op
            // but private ones (protected by a gatekeeper) coul
            // because it would ultimately result in governance 
            // is unattributed/unclaimable, meaning that party w
            // able to reach 100% consensus.
            totalContributions,
            gateData
        );
```

Then crowdfund was a success, party was created, and Melany, who also participated, per off-chain arrangement has transferred to Bob a `tokenId` with big voting power (say it is `100 ETH` and the majority of voting power):

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernanceNFT.sol#L146-L155

```
/// @inheritdoc ERC721
function safeTransferFrom(address owner, address to, uint256
    public
    override
    onlyDelegateCall
{
    // Transfer voting along with token.
    _transferVotingPower(owner, to, votingPowerByTokenId[tok
    super.safeTransferFrom(owner, to, tokenId);
}
```

[https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernance.sol#L879-L887](https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernance.sol#L879-L887)

```
// Transfers some voting power of `from` to `to`. The total
// their respective delegates will be updated as well.
function _transferVotingPower(address from, address to, uint
    internal
{
    int192 powerI192 = power.safeCastUint256ToInt192();
    _adjustVotingPower(from, -powerI192, address(0));
    _adjustVotingPower(to, powerI192, address(0));
}
```

Bob don't care about his early small contribution and focuses on managing the one that Melany transferred instead as he simply doesn't need the voting power from the initial `0.01 ETH` contribution anymore.

The actual delegate for Bob at the moment is Linda, while his business with Mike is over. So Bob sets her address there, calling `delegateVotingPower(Linda)`:

[https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernance.sol#L448-L454](https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernance.sol#L448-L454)

```
/// @notice Pledge your intrinsic voting power to a new dele
///         the old one (if any).
/// @param delegate The address to delegating voting power t
```

```
    function delegateVotingPower(address delegate) external only
        _adjustVotingPower(msg.sender, 0, delegate);
        emit VotingPowerDelegated(msg.sender, delegate);
    }
```

Now, Mike can unilaterally delegate to himself the whole voting power with `burn(Bob)` as mint() just resets the delegation with the previously recorded value with `_adjustVotingPower(owner, votingPower.safeCastUint256ToInt192(), delegate)`.

## Recommended Mitigation Steps

The issue is that `mint()` always assumes that it is the first operation for the `owner`, which might not always be the case.

Consider not changing the delegate on `mint` if one is set already:

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernanceNFT.sol#L120-L133

```
    function mint(
        address owner,
        uint256 votingPower,
        address delegate
    )
        external
        onlyMinter
        onlyDelegateCall
    {
        uint256 tokenId = ++tokenCount;
        votingPowerByTokenId[tokenId] = votingPower;
+       address actualDelegate = <get_current_delegate>;
+       if (actualDelegate == address(0)) actualDelegate = deleg
-       _adjustVotingPower(owner, votingPower.safeCastUint256ToI
+       _adjustVotingPower(owner, votingPower.safeCastUint256ToI
        _mint(owner, tokenId);
    }
```

More complicated version might be the one with tracking the most recent request via `contribute()/delegateVotingPower()` calls timestamps. Here we assume that the `delegateVotingPower()` holds more information as in the majority of practical cases it occurs after initial `contribute()` and it is a direct voluntary call from the owner.

**merklejerk (PartyDAO) confirmed, but disagreed with severity and commented:**

> Unusual and interesting, but we think this should be Medium risk since the setup is somewhat exotic and still not a guarantee that assets would be at risk. We will implement the fix suggested.

**HardlyDifficult (judge) decreased severity to Medium and commented:**

> Agree with downgrading to Medium risk. This scenario can result in redirecting voting power to the wrong account, but is a nuanced scenario that would be difficult to target as an attack.

**0xble (PartyDAO) resolved:**

> Resolved: https://github.com/PartyDAO/partybidV2/pull/134

## [M-03] Only part of `keccak256()` is used as hash, making it susceptible to collision attacks

*Submitted by 0xA5DF, also found by Lambda and V\B_*

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/Crowdfund.sol#L275

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/Crowdfund.sol#L325

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/distribution/TokenDistributor.sol#L26

## Vulnerability Details

At 2 places in the code only part of the output of `keccak256()` is used as the hash:

- At `TokenDistributor - DistributionState.distributionHash15` - uses only a 15 bytes as a hash
    - This one is intended to save storage
- At `Crowdfund.governanceOptsHash` a 16 bytes is used as hash
    - This one has no benefit at all as it doesn't save on storage

15/16 bytes hash is already not very high to begin with (as explained below). On top of that, using a non standard hash can be unsafe. Since diverging from the standard can break things.

## Impact

For the `FixedGovernanceOpts` an attacker can create a legitimate party, and then when running `buy()` use the malicious hash to:

- include himself in the hosts (DoS-ing the party by vetoing every vote)
- reduce the `passThresholdBps` (allowing him to pass any vote, including sending funds from the Party)
- Setting himself as `feeRecipient` and increasing the fee

For the `DistributionInfo` struct - an attacker can easily drain all funds from the token distribution contract, by using the legitimate hash to create a distribution with a malicious ERC20 token (and a malicious party contract), and then using the malicious hash to claim assets of a legitimate token.

## Proof of Concept

### The attack

Using the birthday attack, for a 50% chance with a 15 bytes hash, the number of hashes needed to generate is 1.4e18 ( `(ln(1/0.5) *2) ** 0.5 * (2 ** 60)` ).

- For 16 bytes that would be 2.2e19

An attacker can create 2 different structs, a legitimate and a malicious one, while modifying at each iteration only the last bits

- For the `FixedGovernanceOpts` the last bits would be the `feeRecipient` field
- For the `DistributionInfo` struct that would be the `fee` field (and then exploit it via the `claim()` function which doesn't validate the `fee` field)

The attacker will than generate half of the hashes from the malicious one, and half from the legitimate ones, so in case of a collision there's a 50% chance it'd be between the legitimate and malicious struct.

### CPU

- In the `DistributionInfo` we have 224 bytes (and for `FixedGovernanceOpts` 192 bytes if we precalculate the hosts hash)
- A computer needs about 11 cycles per byte
- An avg home PC can do ~3e9 cycles per seconds
- There are ~8.6e4 seconds a day
- Putting it all together `1.4e18 * 11 * 224 / (3e9*8.6e4)` = ~1.3e8
- Note that we can further optimize it (by 10 times at least), since we're using the same input and only modifying the last bits every time (the `fee` field)

### Storage

32 * 1.4e18 = ~4.5e19 bytes is needed, while an affordable drive can be 8TB=~8e12 bytes. That puts it about 5e6 times away from and affordable attack.

### Overall Risk

The calculations above are for basic equipment, an attacker can be spending more on equipment to get closer (I'd say you can easily multiply that by 100 for a medium size attacker + running the computations for more than one day). Combining that with the fact that a non-standard hash is used, and that in general hashes can have small vulnerabilities that lower a bit their strength - I'd argue it's not very safe to be ~1e4 (for a medium size attacker; ~1.5e5 for 16 bytes) away from a practical attack.

### Recommended Mitigation Steps

Use the standard, 32-bytes, output of `keccak256()`.

[merklejerk (PartyDAO) confirmed and commented](#):

> At first we dismissed this as being really impractical but the more we thought of what an attack would actually look like, the more practical (if still improbable) it would be. We will extend both hashes to full-width (32 bytes).

[HardlyDifficult (judge) commented](#):

> Agree with Medium risk. This attack seems unlikely but may be within the realm of possible when high value is at stake.

[0xble (PartyDAO) resolved](#):

> Resolved: https://github.com/PartyDAO/partybidV2/pull/138

## [M-04] NFT Owner can stuck Crowdfund user funds

*Submitted by csanuragjain*

Consider a scenario where few users contributed in auction but no one has placed any bid due to reason like NFT price crash etc. So there was 0 bid, the NFT owner could seize the crowdfund users fund until they pay a ransom amount as shown below.

### Proof of Concept

1. NFT N auction is going on
2. CrowdFund users have contributed 100 amount for this auction
3. Bidding has not been done yet
4. A news came for this NFT owner which leads to crashing of this NFT price
5. CrowdFund users are happy that they have not bided and are just waiting for auction to complete so that they can get there refund
6. NFT owner realizing this blackmails the CrowdFund users to send him amount 50 or else he would send this worthless NFT to the Crowdfund Auction contract

basically stucking all crowdfund users fund. CrowdFund users ignore this and simply wait for auction to end

7. Once auction completes [finalize function]() is called

```
function finalize(FixedGovernanceOpts memory governanceOpts)
        external
        onlyDelegateCall
        returns (Party party_)
    {
...
 if (nftContract.safeOwnerOf(nftTokenId) == address(this)) {
            if (lastBid_ == 0) {
                // The NFT was gifted to us. Everyone who contril
                lastBid_ = totalContributions;
                if (lastBid_ == 0) {
                    // Nobody ever contributed. The NFT is effec
                    revert NoContributionsError();
                }
                lastBid = lastBid_;
            }
            // Create a governance party around the NFT.
            party_ = _createParty(
                _getPartyFactory(),
                governanceOpts,
                nftContract,
                nftTokenId
            );
            emit Won(lastBid_, party_);
        }
...
}
```

8. Before calling finalize the lastBid was 0 since no one has bid on this auction but lets see what happens on calling finalize
9. Since NFT owner has transferred NFT to this contract so below statement holds true and `lastBid\_` is also 0 since no one has bidded

```
if (lastBid_ == 0) {
                lastBid_ = totalContributions;
```

10. This means now `lastBid\_` is changed to totalContributions which is 100 so crowdfund users funds will not be refunded and they will end up with non needed NFT.

## Recommended Mitigation Steps

Remove the line `lastBid\_ = totalContributions;` and let it be the last bid amount which crowdfund users actually bided with.

**merklejerk (PartyDAO) disputed and commented:**

> This is working as designed. Contributors to a crowdfund essentially enter an agreement to pay up to a maximum price of their combined contributions for an NFT.

**csanuragjain (warden) commented:**

> @merklejerk But in this case it is not "up to a maximum price" but rather always the maximum price using the PoC.

**merklejerk (PartyDAO) confirmed and commented:**

> Hmm. I reread the PoC and now I'm converting this to confirmed. The real issue here is not so much that the party is paying maximum price, but that the party did not ever bid on the item but also paid maximum price. :facepalm:

**merklejerk (PartyDAO) disagreed with severity and commented:**

> Still disagree with severity since this is unlikely to happen with a legitimate collection. It should be Medium at most.

**HardlyDifficult (judge) decreased severity to Medium and commented:**

> Agree with Medium risk here. This could be a violation of party user expectations since auctions generally target the min possible bid - a form of leaking value.

**0xble (PartyDAO) resolved:**

> Resolved: https://github.com/PartyDAO/partybidV2/pull/133

## [M-05] The settledPrice maybe exceed maximumPrice

*Submitted by bin2chen*

`BuyCrowdfundBase.sol \_buy()` When callValue = 0 is settledPrice to totalContributions ignoring whether totalContributions > maximumPrice resulting in the minimum proportion of participants expected to become smaller

### Proof of Concept

```
        function _buy(
            IERC721 token,
            uint256 tokenId,
            address payable callTarget,
            uint96 callValue,
            bytes calldata callData,
            FixedGovernanceOpts memory governanceOpts
        )
        ...
                settledPrice_ = callValue == 0 ? totalContributions
                if (settledPrice_ == 0) {
                    // Still zero, which means no contributions.
                    revert NoContributionsError();
                }
                settledPrice = settledPrice_;
```

(AuctionCrowdfund.sol finalize() similar)

### Recommended Mitigation Steps

add check

```
        function _buy(
            IERC721 token,
            uint256 tokenId,
            address payable callTarget,
            uint96 callValue,
            bytes calldata callData,
            FixedGovernanceOpts memory governanceOpts
        )
        ...
                settledPrice_ = callValue == 0 ? totalContributions
                if (settledPrice_ == 0) {
                    // Still zero, which means no contributions.
                    revert NoContributionsError();
                }

    +++         if (maximumPrice_ != 0 && settledPrice_ > maximumPri
    +++             settledPrice_ = maximumPrice_;
    +++         }

                settledPrice = settledPrice_;
```

[merklejerk (PartyDAO) confirmed and commented](#):

> We will cap the callValue to maximumPrice.

[0xble (PartyDAO) resolved](#):

> Resolved: [https://github.com/PartyDAO/partybidV2/pull/133](https://github.com/PartyDAO/partybidV2/pull/133)

[HardlyDifficult (judge) commented](#):

> This is a potential violation of user expectations - agree with Medium risk.

## [M-06] AuctionCrowdfund: If the contract was bid on before the NFT was gifted to the contract, lastBid will not be totalContributions

*Submitted by cccz*

In the finalize function of the AuctionCrowdfund contract, when the contract gets NFT and lastBid_ == 0, it is considered that NFT is gifted to the contract and everyone who contributed wins.

```
            if (nftContract.safeOwnerOf(nftTokenId) == address(this)
                if (lastBid_ == 0) {
                    // The NFT was gifted to us. Everyone who contri
```

```
lastBid_ = totalContributions;
```

But if the contract was bid before the NFT was gifted to the contract, then since lastBid_ ! = 0, only the user who contributed at the beginning will win.

### Proof of Concept

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/AuctionCrowdfund.sol#L233-L242

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/AuctionCrowdfund.sol#L149-L175

### Recommended Mitigation Steps

Whether or not NFT is free to get should be determined using whether the contract balance is greater than totalContributions.

merklejerk (PartyDAO) confirmed and commented:

> Gifted NFTs are expected to be an extremely exceptional (if ever) case, but we do want to make a best effort to make these situations somewhat fair to contributors. We will implement the recommendation and check that `this.balance >= totalContributions` to see if the NFT was acquired for free.

0xble (PartyDAO) resolved:

> Resolved: https://github.com/PartyDAO/partybidV2/pull/133

HardlyDifficult (judge) commented:

> This could be seen as a form of leaking value, agree with Medium risk.

## [M-07] Attacker can force AuctionCrowdfunds to bid their entire contribution up to maxBid

*Submitted by Trust, also found by cccz*

AuctionCrowdfund's `bid()` allows any user to compete on an auction on the party's behalf. The code in `bid()` forbids placing a bid if party is already winning the auction:

```
    if (market.getCurrentHighestBidder(auctionId_) == address(this))
                revert AlreadyHighestBidderError();
            }
```

However, it does not account for attackers placing bids from their own wallet, and then immediately overbidding them using the party's funds. This can be used in two ways:

1. Attacker which lists an NFT, can force the party to spend all its funds up to maxBid on the auction, even if the party could have purchased the NFT for much less.
2. Attackers can grief random auctions, making them pay the absolute maximum for the item. Attackers can use this to drive the prices of NFT items up, profiting from this using secondary markets.

## Impact

Parties can be stopped from buying items at a good value without any risk to the attacker.

## Proof of Concept

1. Attacker places an NFT for sale, valued at X
2. Attacker creates an AuctionCrowdfund, with maxBid = Y such that Y = 2X
3. Current bid for the NFT is X - AUCTION_STEP
4. Users contribute to the fund, which now has 1.5X
5. Users call `bid()` to bid X for the NFT
6. Attacker bids for the item externally for 1.5X - AUCTION_STEP
7. Attacker calls `bid()` to bid 1.5X for the NFT

8. Attacker sells his NFT for 1.5X although no one apart from the party was interested in buying it above price X

## Recommended Mitigation Steps

Introduce a new option variable to AuctionCrowdfunds called speedBump. Inside the `bid()` function, calculate seconds since last bid, multiplied by the price change factor. This product must be smaller than the chosen speedBump. Using this scheme, the protocol would have resistance to sudden bid spikes. Optionally, allow a majority funder to override the speed bump.

**merklejerk (PartyDAO) acknowledged and commented:**

> This is a known limitation of crowdfunds. We will allow some parties to restrict who can call `buy()` or `bid()` to hosts, which will mitigate this.

**HardlyDifficult (judge) commented:**

> This is a fair concern, a form of potentially leaking value so agree with Medium risk. Not sure I agree with the recommendation here, but restricting to hosts does help mitigate by putting risk on the attacker.

**0xble (PartyDAO) confirmed and resolved:**

> Mitigated by: https://github.com/PartyDAO/partybidV2/pull/140

## [M-08] Early contributor can always become majority of crowdfund leading to rugging risks.

*Submitted by Trust, also found by cccz and rvierdiiev*

Voting power is distributed to crowdfund contributors according to the amount contributed divided by NFT purchase price. Attacker can call the `buy()` function of BuyCrowdfund / CollectionBuyCrowdfund, and use only the first X amount of contribution from the crowdfund, such that attacker's contribution > X/2. He will pass his contract to the buy call, which will receive X and will need to add some additional funds, to purchase the NFT. If the purchase is successful, attacker will have majority rule in the created party. If the party does not do anything malicious, this is a losing move for attacker, because the funds they added on top of X to compensate for the NFT price will eventually be split between group members. However, with majority rule there are various different exploitation vectors attacker may use to steal the NFT from the party ( detailed in separate reports). Because it is accepted that single actor majority is dangerous, but without additional vulnerabilities attacker cannot take ownership of the party's assets, I classify this as a Medium. The point is that users were not aware they could become minority under this attack flow.

## Impact

Early contributor can always become majority of crowdfund leading to rugging risks.

## Proof of Concept

1. Victim A opens BuyCrowdfund and deposits 20 ETH
2. Attacker deposits 30 ETH
3. Victim B deposits 50 ETH
4. Suppose NFT costs 100 ETH
5. Attacker will call `buy()` , requesting 59ETH buy price. His contract will add 41 additional ETH and buy the NFT.
6. Voting power distributed will be: 20 / 59 for Victim A, 30 / 59 for Attacker, 9 / 59 for Victim B. Attacker has majority.
7. User can use some majority attack to take control of the NFT, netting 100 (NFT value) - 41 (external contribution) - 30 (own contribution) = 29 ETH

## Recommended Mitigation Steps

Add a Crowdfund property called minimumPrice, which will be visible to all. `Buy()` function should not accept NFT price < minimumPrice. Users now have assurances that are not susceptible to majority rule if they deposited enough ETH below the minimumPrice.

[merklejerk (PartyDAO) acknowledged and commented](#):

> We want to avoid requiring a minimum price for now as it's difficult to choose a value that would allow a crowdfund to respond to price drops as well as price increases. Instead we're opting to harden defenses against majority attacks, which would minimize the desirability of this vector, and we're introducing optional host-

> only and gatekeeper restrictions on `buy()/bid()` functions. We will keep an eye out for this behavior and reassess if it comes up.

[HardlyDifficult (judge) commented]:

> Gifting can lead to incorrect voting power distribution. Agree with Medium risk.

[0xble (PartyDAO) confirmed and resolved]:

> Mitigated by: https://github.com/PartyDAO/partybidV2/pull/140

## [M-09] Calling `transferEth` function can revert if `receiver` input corresponds to a contract that is unable to receive ETH through its `receive` or `fallback` function

*Submitted by rbserver, also found by CertoraInc, Jeiwan, and tonisives*

https://github.com/PartyDAO/party-contracts-c4/blob/main/contracts/utils/LibAddress.sol#L8-L15

https://github.com/PartyDAO/party-contracts-c4/blob/main/contracts/crowdfund/Crowdfund.sol#L444-L489

https://github.com/PartyDAO/party-contracts-c4/blob/main/contracts/distribution/TokenDistributor.sol#L371-L388

### Impact

The following `transferEth` function is called when calling the `_burn` or `_transfer` function below. If the `receiver` input for the `transferEth` function corresponds to a contract, it is possible that the receiver contract does not, intentionally or unintentionally, implement the `receive` or `fallback` function in a way that supports receiving ETH or that calling the receiver contract's `receive` or `fallback` function executes complicated logics that cost much gas, which could cause calling `transferEth` to revert. For example, when calling `transferEth` reverts, calling `_burn` also reverts; this means that the receiver contract would not be able to get the voting power and receive the extra contribution it made after the crowdfunding finishes; yet, the receiver contract deserves these voting power and contribution refund. Hence, the receiver contract loses valuables that it deserves, which is unfair to the users who controls it.

https://github.com/PartyDAO/party-contracts-c4/blob/main/contracts/utils/LibAddress.sol#L8-L15

```
function transferEth(address payable receiver, uint256 amoun
    internal
{
    (bool s, bytes memory r) = receiver.call{value: amount}(
    if (!s) {
        revert EthTransferFailed(receiver, r);
    }
}
```

https://github.com/PartyDAO/party-contracts-c4/blob/main/contracts/crowdfund/Crowdfund.sol#L444-L489

```
function _burn(address payable contributor, CrowdfundLifecyc
    private
{
    // If the CF has won, a party must have been created pri
    if (lc == CrowdfundLifecycle.Won) {
        if (party_ == Party(payable(0))) {
            revert NoPartyError();
        }
    } else if (lc != CrowdfundLifecycle.Lost) {
        // Otherwise it must have lost.
        revert WrongLifecycleError(lc);
    }
    // Split recipient can burn even if they don't have a to
    if (contributor == splitRecipient) {
        if (_splitRecipientHasBurned) {
            revert SplitRecipientAlreadyBurnedError();
        }
        _splitRecipientHasBurned = true;
    }
    // Revert if already burned or does not exist.
    if (splitRecipient != contributor || _doesTokenExistFor(
        CrowdfundNFT._burn(contributor);
    }
    // Compute the contributions used and owed to the contril
```

```
        // with the voting power they'll have in the governance 
        (uint256 ethUsed, uint256 ethOwed, uint256 votingPower) 
            _getFinalContribution(contributor);
        if (votingPower > 0) {
            // Get the address to delegate voting power to. If n
            address delegate = delegationsByContributor[contribu
            if (delegate == address(0)) {
                // Delegate can be unset for the split recipient
                // contribute. Self-delegate if this occurs.
                delegate = contributor;
            }
            // Mint governance NFT for the contributor.
            party_.mint(
                contributor,
                votingPower,
                delegate
            );
        }
        // Refund any ETH owed back to the contributor.
        contributor.transferEth(ethOwed);
        emit Burned(contributor, ethUsed, ethOwed, votingPower);
    }
```

[https://github.com/PartyDAO/party-contracts-c4/blob/main/contracts/distribution/TokenDistributor.sol#L371-L388](https://github.com/PartyDAO/party-contracts-c4/blob/main/contracts/distribution/TokenDistributor.sol#L371-L388)

```
    function _transfer(
        TokenType tokenType,
        address token,
        address payable recipient,
        uint256 amount
    )
        private
    {
        bytes32 balanceId = _getBalanceId(tokenType, token);
        // Reduce stored token balance.
        _storedBalances[balanceId] -= amount;
        if (tokenType == TokenType.Native) {
            recipient.transferEth(amount);
        } else {
            assert(tokenType == TokenType.Erc20);
            IERC20(token).compatTransfer(recipient, amount);
        }
```

```
}
```

## Proof of Concept

Please add the following `error` and append the test in `sol-tests\crowdfund\BuyCrowdfund.t.sol`. This test will pass to demonstrate the described scenario.

```
error EthTransferFailed(address receiver, bytes errData);

function testContributorContractFailsToReceiveETH() public {
    uint256 tokenId = erc721Vault.mint();
    BuyCrowdfund pb = _createCrowdfund(tokenId, 0);

    // This contract is used to simulate a contract that doe
    address payable contributorContract = payable(address(th
    vm.deal(contributorContract, 1e18);

    address delegate = _randomAddress();

    // contributorContract contributes 1e18.
    vm.prank(contributorContract);
    pb.contribute{ value: 1e18 }(delegate, "");

    // The price of the NFT of interest is 0.5e18.
    Party party_ = pb.buy(
        payable(address(erc721Vault)),
        0.5e18,
        abi.encodeCall(erc721Vault.claim, (tokenId)),
        defaultGovernanceOpts
    );

    // After calling the buy function, the party is created
    assertEq(address(party), address(party_));
    assertTrue(pb.getCrowdfundLifecycle() == Crowdfund.Crowd
    assertEq(pb.settledPrice(), 0.5e18);
    assertEq(pb.totalContributions(), 1e18);
    assertEq(address(pb).balance, 1e18 - 0.5e18);

    // Calling the burn function reverts because contributor
    vm.expectRevert(abi.encodeWithSelector(
        EthTransferFailed.selector,
        contributorContract,
        ""
```

```
            ));
            pb.burn(contributorContract);

            // contributorContract does not receive 0.5e18 back from
            assertEq(contributorContract.balance, 0);
        }
```

### Tools Used

VSCode

### Recommended Mitigation Steps

When calling the `transferEth` function, if the receiver contract is unable to receive ETH through its `receive` or `fallback` function, WETH can be used to deposit the corresponding ETH amount, and the deposited amount can be transferred to the receiver contract.

**merklejerk (PartyDAO) confirmed and commented:**

> We will mitigate this by escrowing the ETH refund and/or the governance NFT to be claimed by the contributor later (possibly to a different address) if either transfer fails.

**0xble (PartyDAO) resolved:**

> Resolved: https://github.com/PartyDAO/partybidV2/pull/126

**HardlyDifficult (judge) commented:**

> A contract contributor may revert on `burn` , preventing full voting power distribution. Agree with Medium risk.

## [M-10] Possible that unanimous votes is unachievable

*Submitted by Lambda, also found by CertoraInc*

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/crowdfund/Cr owdfund.sol#L370

https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernance.sol#L1066

## Impact

Currently, the `votingPower` calculation rounds down for every user except the `splitRecipient`. This can cause situations where 99.99% of votes (i.e., an unanimous vote) are not achieved, even if all vote in favor of a proposal. This can be very bad for a party, as certain proposals (transferring precious tokens out) require an unanimous vote and are therefore not executable.

## Proof Of Concept

Let's say for the sake of simplicity that 100 persons contribute 2 wei and `splitBps` is 10 (1%). `votingPower` for all users that contributed will be 1 and 2 for the `splitRecipient`, meaning the maximum achievable vote percentage is 102 / 200 = 51%.

Of course, this is an extreme example, but it shows that the current calculation can introduce siginificant rounding errors that impact the functionality of the protocol.

## Recommended Mitigation Steps

Instead of requiring more than 99.99% of the votes, ensure that the individual votingPower sum to the total contribution. For instance, one user (e.g., the last one to claim) could receive all the remaining votingPower, which would require a running sum of the already claimed votingPower.

merklejerk (PartyDAO) acknowledged and commented:

> We consider it highly unlikely that an NFT would be crowdfunded for dust amounts so we don't think this would actually occur in the wild.

HardlyDifficult (judge) commented:

> Rounding error could prevent unanimous votes.. agree with Medium risk.

## [M-11] Maximum bid will always be used in Auction

*Submitted by csanuragjain, also found by Lambda*

AuctionCrowdfund contract is designed in a way to allow bidding max upto maximumBid. But due to a flaw, anyone (including NFT seller) can make sure that CrowdFund bid always remain equal to maximumBid thus removing the purpose of maximumBid. This also causes loss to Party participating in this Auction as the auction will always end up with maximumBid even when it could have stopped with lower bid as shown in POC.

### Proof of Concept

1. An auction is started for NFT N in the market
2. Party Users P1 starts an AuctionCrowdfund with maximumBid as 100 for this auction.

```
function initialize(AuctionCrowdfundOptions memory opts)
        external
        payable
        onlyConstructor
    {
...
maximumBid = opts.maximumBid;
...
}
```

3. P1 bids amount 10 for the NFT N using bid function
4. Some bad news arrives for the NFT collection including NFT N reducing its price
5. P1 decides not to bid more than amount 10 due to this news
6. NFT collection owner who is watching this AuctionCrowdfund observes that bidding is only 10 but Party users have maximumBid of 100
7. NFT collection owner asks his friend to bid on this NFT in the auction market (different from crowd fund)
8. NFT collection owner now takes advantage of same and himself calls the bid function of AuctionCrowdfund via Proxy

```
    function bid() external onlyDelegateCall {
    ...
    }
```

9. Now since last bid belongs to collection owner friend, so AuctionCrowdfund contract simply extends its bid further

```
    if (market.getCurrentHighestBidder(auctionId_) == address(this))
                revert AlreadyHighestBidderError();
            }
            // Get the minimum necessary bid to be the highest bidde
            uint96 bidAmount = market.getMinimumBid(auctionId_).safe
            // Make sure the bid is less than the maximum bid.
            if (bidAmount > maximumBid) {
                revert ExceedsMaximumBidError(bidAmount, maximumBid)
            }
            lastBid = bidAmount;
```

10. NFT collection owner keeps repeating step 7-9 until AuctionCrowdfund reaches the final maximum bid of 100
11. After auction completes, collection owner gets 100 amount instead of 10 even though crowd fund users never bidded for amount 100

## Recommended Mitigation Steps

maximumbid concept can easily be bypassed as shown above and will not make sense. Either remove it completely

OR

bid function should only be callable via crowdfund members then attacker would be afraid if new bid will come or not and there should be a consensus between crowdfund members before bidding which will protect this scenario.

[merklejerk (PartyDAO) acknowledged and commented](#):

> While it doesn't solve it for all crowdfunds, we will allow some crowdfunds to restrict who can call their `bid()` function to host-only.

HardlyDifficult (judge) commented:

> This is a form of leaking value - agree with Medium risk.

0xble (PartyDAO) confirmed and resolved:

> Mitigated by: https://github.com/PartyDAO/partybidV2/pull/140

## [M-12] Excess eth is not refunded

*Submitted by csanuragjain*

The ArbitraryCallsProposal contract requires sender to provide eth(msg.value) for each call. Now if user has provided more eth than combined call.value then this excess eth is not refunded back to user.

### Proof of Concept

1. Observe the _executeArbitraryCalls function

```
function _executeArbitraryCalls(
        IProposalExecutionEngine.ExecuteProposalParams memory pa
    )
        internal
        returns (bytes memory nextProgressData)
    {

...
uint256 ethAvailable = msg.value;
        for (uint256 i = 0; i < calls.length; ++i) {
            // Execute an arbitrary call.
            _executeSingleArbitraryCall(
                i,
                calls[i],
                params.preciousTokens,
                params.preciousTokenIds,
                isUnanimous,
                ethAvailable
            );
            // Update the amount of ETH available for the subseq
            ethAvailable -= calls[i].value;
```

```
                emit ArbitraryCallExecuted(params.proposalId, i, cal
            }
    ....
    }
```

2. As we can see user provided msg.value is deducted with each calls[i].value
3. Assume user provided 5 amount as msg.value and made a single call with calls[0].value as 4
4. This means after calls have been completed ethAvailable will become 5-4=1
5. Ideally this 1 eth should be refunded back to user but there is no provision for same and the fund will remain in contract

### Recommended Mitigation Steps

At the end of `\_executeArbitraryCalls` function, refund the remaining ethAvailable back to the user.

**merklejerk (PartyDAO) confirmed and commented:**

> Will refund unused ETH at the end of executing arbitrary calls.

**0xble (PartyDAO) resolved:**

> Resolved: https://github.com/PartyDAO/partybidV2/pull/135

**HardlyDifficult (judge) commented:**

> This is a form of leaking value - agree with Medium risk.

## Low Risk and Non-Critical Issues

For this contest, 67 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Lambda** received the top score from the judge.

*The following wardens also submitted reports:* Ch_301, 0xSmartContract, Chom, brgltd, 0xNazgul, ayeslick, 0x52, CodingNameKiki, CertoraInc, cryptphi, Trust, c3phas, Deivitto, R2, pfapostol, CRYP70, 0xbepresent, 0x1f8b, asutorufos, rvierdiiev, bulej93, hansfriese, ladboy233, RaymondFam, gogo, leosathya, pedr02b2, MiloTruck, csanuragjain, fatherOfBlocks, wagmi, bin2chen, lukris02, KIntern_NA, Funen, ReyAdmirado, 0x5rings, Jeiwan, PaludoX0, ChristianKuri, smiling_heretic, __141345__, delfin454000, slowmoses, malinariy, cccz, Aymen0909, martin, The_GUILD, Anth3m, 0xDanielC, erictee, 0x4non, Tomo, JansenC, MasterCookie, djxploit, ch0bu, Olivierdem, tnevler, B2, StevenL, JC, V_B, indijanc, *and* d3e4.

## Low Risk Issues

- After a `FractionalizeProposal` was executed succesfully and the tokens were distributed, it will not be possible to get the NFT back by calling `redeem` in practice. This function requires that the sender (the party in this case) owns all tokens, which would require that all users transfer them back again. However, transferring them to the party would be very risky when you do not if all other participants also do that (as the tokens are lost when one user does not transfer them) and some users may have sold them. This behavior may be desired (although it is a bit against the general philosophy were the tokens are protected IMO), but it should be documented that this is a destructive action.
- `FoundationMarketWrapper.auctionIdMatchesToken` returns `true` for auctionId 0 and `isFinalized` also returns `true`. Because of that, it is possible to create an auction with ID 0 where no one can bid because it is immediately finalized.
- `ZoraMarketWrapper.auctionIDMatchesToken` returns `true` for auctionId 0 and `nftContract = address(0)`, and `isFinalized` als returns `true`. Because of that, it is possible to create an auction with ID 0 and `address(0)` where no one can bid because it is immediately finalized.
- `BuyCrowdfund` cannot retrieve ETH, but certain contracts that are called by it (e.g., NFT marketplaces) might return additional ETH when too much is paid. In such scenarios, the whole transaction would fail.

## Non-Critical Issues

- In `PartyGovernance._getProposalStatus`, the constant `VETO_VALUE` is not used (https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/Part yGovernance.sol#L1033), whereas it is used when performing the veto (https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/party/PartyGovernance.sol#L634). While this is not a problem currently (the values are identical), it can be dangerous when this value is updated at one point, because updating `_getProposalStatus` might get forgotten. Consider using the constant consistently.

- The link for FractionalizeProposal (https://github.com/PartyDAO/party-contracts-c4/blob/3896577b8f0fa16cba129dc2867aba786b730c1b/contracts/proposals/FractionalizeProposal.sol#L30) is wrong. It links to the source code of the vault, but the variable points to the factory. The correct link is https://github.com/fractional-company/contracts/blob/master/src/ERC721VaultFactory.sol
- For Nouns, it is only possible to deploy the auction when the desired ID is for sale. However, it might be desired to create an auction in advance (e.g., for token ID 7 when the current ID is 5), such that there is more time to collect funds. For a system like Nouns this would work nicely, because it is (roughly) known in advance when an auction for a token will start.
- `CrowdfundNFT`, the implementation of a soulbound NFT, is a bit too simplistic in my opinion. One problem with soulbound NFTs is wallet recovery. In such situations, you want to be able to transfer the NFT to another wallet (but of course, this should not be possible without prior checks, otherwise they would not be soulbound). There are different standards like EIP 4671 that address soulbound NFTs (and the mentioned problems), consider implementing one of them.

0xble (PartyDAO) commented:

> The first two Non-Critical suggestions we'll change, the rest are interesting and we acknowledge them. Great suggestions overall.

HardlyDifficult (judge) commented:

> Merging with 109, 103, 106, 108, 115, and 116 which are all Low Risk issues.

## Gas Optimizations

For this contest, 82 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **CertoraInc** received the top score from the judge.

*The following wardens also submitted reports:* m_Rassska, pfapostol, c3phas, JC, Deivitto, MiloTruck, ReyAdmirado, 0xSmartContract, gogo, Rolezn, 0x1f8b, 0xkatana, JAGADESH, Sm4rty, ajtra, peiw, SnowMan, brgltd, erictee, prasantgupta52, Rohan16, Bnke0x0, __141345__, Aymen0909, gianganhnguyen, karanctf, Tomio, sryysryy, 0xNazgul, ignacio, lukris02, RaymondFam, Lambda, malinariy, ch0bu, Metatron, Waze, 0x5rings, djxploit, fatherOfBlocks, leosathya, robee, Saintcode_, jag, martin, slowmoses, Tomo, got_targ, Olivierdem, 0x4non, Amithuddar, asutorufos, B2, Chom, ChristianKuri, CRYP70, delfin454000, Fitraldys, ladboy233, LeoS, natzuu, simon135, tnevler, V_B, bulej93, Diraco, francoHacker, Noah3o6, Ocean_Sky, d3e4, dharma09, IgnacioB, peanuts, 0x85102, aysha, CodingNameKiki, Funen, Matin, PaludoX0, pashov, *and* StevenL.

- Use the old `oldHostsFieldValue` value in the `Crowdfund._hashFixedGovernanceOpts`

  ```
  let oldHostsFieldValue := mload(opts)
  mstore(opts, keccak256(add(mload(opts), 0x20), mul(mload(mload(opts)),
  ```

- Use unchecked on this part of the `Crowdfund._getFinalContribution` function:

  ```
  ...
  } else {
      // This contribution was partially used.
      uint256 partialEthUsed = totalEthUsed - c.previousTotalContribution
      ethUsed += partialEthUsed; // this doesn't need to be unchecked
      ethOwed = c.amount - partialEthUsed;
  }
  ```

  The code will look like this:

  ```
  ...
  } else {
  ```

```
        // This contribution was partially used.
        unchecked {
            uint256 partialEthUsed = totalEthUsed - c.previousTotalContribu
            ethOwed = c.amount - partialEthUsed;
        }
        ethUsed += partialEthUsed; // this doesn't need to be unchecked
    }
```

- Use > 0 instead of >= 1, use unchecked and cache the expression's result instead of calculating it twice (in the `Crowdfund._contribute` function):

```
    if (numContributions >= 1) {
        Contribution memory lastContribution = contributions[numContributio
        if (lastContribution.previousTotalContributions == previousTotalCon
            // No one else has contributed since so just reuse the last ent
            lastContribution.amount += amount;
            contributions[numContributions - 1] = lastContribution;
            return;
        }
    }
```

  The code will look like this:

```
    if (numContributions > 0) {
        unchecked {
            uint lastIndex = numContributions - 1;
        }
        Contribution memory lastContribution = contributions[lastIndex];
        if (lastContribution.previousTotalContributions == previousTotalCon
            // No one else has contributed since so just reuse the last ent
            lastContribution.amount += amount;
            contributions[lastIndex] = lastContribution;
            return;
        }
    }
```

- Cache `splitRecipient` in the `Crowdfund._burn()` function
- If the amount is 0, don't perform the low level call in `LibAddress.transferETH()`

## Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

Top