Party Protocol - Versus contest Findings & Analysis Report

2023-06-23

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 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 outlined in this document, C4 conducted an analysis of the Party Protocol smart contract system written in Solidity. The audit took place between April 3—April 14 2023.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 4 wardens contributed reports:

1. 0x52
2. HollaDieWaldfee
3. evan
4. hansfriese

This audit was judged by 0xean.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities. Of these vulnerabilities, 8 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 4 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 2 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 Party Protocol audit repository, and is composed of 3 abstracts and 11 smart contracts written in the Solidity programming language and includes 2,570 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (8)

[H-01] Self-delgated users can have their delegation unknowingly hijacked during crowdfunding

Submitted by 0x52

Self-delegation can be hijacked.

Proof of Concept

PartyGovernance.sol#L886-L906

```
function _adjustVotingPower(address voter, int192 votingPower, a
    VotingPowerSnapshot memory oldSnap = _getLastVotingPowerSnap
    address oldDelegate = delegationsByVoter[voter];
    // If `oldDelegate` is zero and `voter` never delegated, the
    // `voter` delegate to themself.
    oldDelegate = oldDelegate == address(0) ? voter : oldDelegat
    // If the new `delegate` is zero, use the current (old) dele
    delegate = delegate == address(0) ? oldDelegate : delegate;

    VotingPowerSnapshot memory newSnap = VotingPowerSnapshot({
        timestamp: uint40(block.timestamp),
        delegatedVotingPower: oldSnap.delegatedVotingPower,
        intrinsicVotingPower: (oldSnap.intrinsicVotingPower.safe
            votingPower).safeCastInt192ToUint96(),
        isDelegated: delegate != voter
    });
    _insertVotingPowerSnapshot(voter, newSnap);
    delegationsByVoter[voter] = delegate;
    // Handle rebalancing delegates.
    _rebalanceDelegates(voter, oldDelegate, delegate, oldSnap, r
}
```

Self-delegation is triggered when a user specifies their delegate as address(0). This means that if a user wishes to self-delegate they will can contribute to a crowdfund with delegate == address(0).

[ETHCrowdfundBase.sol#L169-L181]

```
function _processContribution(
    address payable contributor,
    address delegate,
    uint96 amount
) internal returns (uint96 votingPower) {
    address oldDelegate = delegationsByContributor[contributor];
    if (msg.sender == contributor || oldDelegate == address(0))
        // Update delegate.
        delegationsByContributor[contributor] = delegate;
    } else {
        // Prevent changing another's delegate if already delega
        delegate = oldDelegate;
    }
```

This method of self-delegation is problematic when combined with _processContribution. When contributing for someone else, the caller is allowed to specify any delegate they wish. If that user is currently self delegated, then the newly specified delegate will overwrite their self delegation. This allows anyone to hijack the voting power of a self-delegated user.

This can create serious issues for ReraiseETHCrowdfund because party NFTs are not minted until after the entire crowdfund is successful. Unlike InitialETHCrowdfund, this allows the attacker to hijack all of the user's newly minted votes.

Example:
`minContribution = 1` and `maxContribution = 100`. User A contributes 100 to ReraiseETHCrowdfund. They wish to self-delegate so they call contribute with delegate == address(0). An attacker now contributes 1 on behalf of User A with themselves as the delegate. Now when the NFTs are claimed, they will be delegated to the attacker.

Recommended Mitigation Steps
Self-delegation should be automatically hardcoded:

```
+    if (msg.sender == contributor && delegate == address(0)) {
+        delegationsByContributor[contributor] = contributor;
+    }

     address oldDelegate = delegationsByContributor[contributor];
     if (msg.sender == contributor || oldDelegate == address(0))
         // Update delegate.
         delegationsByContributor[contributor] = delegate;
     } else {
         // Prevent changing another's delegate if already delega
         delegate = oldDelegate;
     }
```

Oxean (judge) commented:

> This appears valid at first pass and allows anyone to steal self delegations.

Oxble (Party) confirmed

[H-02] An attacker can contribute to the ETH crowdfund using a flash loan and control the party as he likes

Submitted by hansfriese

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ETHCrowdfundBase.sol#L273

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/party/PartyGovernance.sol#L470

An attacker can have more than half of the total voting power using a flash loan and abuse other contributors.

Proof of Concept

The main flaw is that the party can distribute funds right after the crowdfund is finalized within the same block.

So the attacker can contribute using a flash loan and repay by distributing the part's ETH.

1. Let's assume `maxContribution = type(uint96).max, minTotalContributions = 10 ether, maxTotalContributions = 20 ether, fundingSplitBps = 0`.
2. An attacker contributes 1 ether(attacker's fund) to the crowdfund and another user contributes 9 ether.
3. The attacker knows the crowdfund will be finalized as it satisfies the `minTotalContributions` already but he will have 10% of the total voting power.
4. So he decides to contribute 10 ether using a flash loan.
5. In `ETHCrowdfundBase._processContribution()`, the crowdfund will be finalized immediately as total contribution is greater than maxTotalContributions.

6. Then the attacker will have `(1 + 10) / 20 = 55%` voting power of the party and he can pass any proposal.
7. So he calls `distribute()` with 19 ether. `distribute()` can be called directly if `opts.distributionsRequireVote == false`, otherwise, he should create/execute the distribution proposal and he can do it within the same block.
8. After that, he can receive ETH using `TokenDistributor.claim()` and the amount will be `19 * 55% = 10.45 ether`. (We ignore the distribution fee for simplicity)
9. He repays 10 ether to the flash loan provider and he can control the party as he likes now.

This attack is possible for both `InitialETHCrowdfund` and `ReraiseETHCrowdfund`.

Recommended Mitigation Steps

I think we should implement a kind of `cooldown logic` after the crowdfund is finalized.

1. Add a `partyStartedTime` in `PartyGovernance.sol`.
2. While finalizing the ETH crowdfund in `ETHCrowdfundBase._finalize()`, we set `party.partyStartedTime = block.timestamp`.
3. After that, `PartyGovernance.distribute()` can work only when `block.timestamp > partyStartTime`.

Oxean (judge) commented:

> Coded POC would have been welcomed here due to the number of steps in the attack, will review further.

hansfriese commented:

> Hello @0xean - Here is a POC. It should be appended to `InitialETHCrowdfund.t.sol`

```
function test_finalizeUsingFlashloan() public {
        InitialETHCrowdfund crowdfund = _createCrowdfund({
```

```
        initialContribution: 0,
        initialContributor: payable(address(0)),
        initialDelegate: address(0),
        minContributions: 0,
        maxContributions: type(uint96).max,
        disableContributingForExistingCard: false,
        minTotalContributions: 10 ether,
        maxTotalContributions: 20 ether,
        duration: 7 days,
        fundingSplitBps: 0,
        fundingSplitRecipient: payable(address(0))
    });

    TokenDistributor distributor = new TokenDistributor(glob
    globals.setAddress(LibGlobals.GLOBAL_TOKEN_DISTRIBUTOR,

    Party party = crowdfund.party();

    // Attacker has 1 ether now
    address attacker = _randomAddress();
    vm.deal(attacker, 1 ether);

    // An honest member has 9 ether
    address member = _randomAddress();
    vm.deal(member, 9 ether);

    // Contribute
    vm.prank(attacker);
    uint256 vp1 = crowdfund.contribute{ value: 1 ether }(att

    vm.prank(member);
    crowdfund.contribute{ value: 9 ether }(member, "");

    // Attacker noticed his voting power will be 10% after f
    vm.deal(attacker, 10 ether); // he borrowed 10 ether

    vm.prank(attacker);
    uint256 vp2 = crowdfund.contributeFor{ value: 10 ether }

    // Crowdfund is finalized
    assertTrue(crowdfund.getCrowdfundLifecycle() == ETHCrowd
    assertEq(party.getGovernanceValues().totalVotingPower, 2
    assertEq(vp1 + vp2, 11 ether); //his voting power is 11/
    assertEq(address(party).balance, 20 ether);
    assertEq(address(attacker).balance, 0); //attacker's eth
```

```
        // attacker starts eth distribution of 19 ether from par
        vm.prank(attacker);
        ITokenDistributor.DistributionInfo memory distInfo = par

        assertEq(address(distributor).balance, 19 ether); //dist
        assertEq(address(party).balance, 1 ether); //party's rem

        vm.prank(attacker);
        distributor.claim(distInfo, 1); //attacker claims 55% of

        assertEq(address(attacker).balance, 10.45 ether); //fina
        assertEq(party.getGovernanceValues().totalVotingPower, 2
        assertEq(vp1 + vp2, 11 ether); //his voting power is sti
    }
```

0xble (Party) confirmed and commented:

> Great finding, still debating the mitigation internally.

0xble (Party) acknowledged and commented:

> Looking into this more, the issue can only occur if a party sets an `executionDelay` of 0. In the POC, the party was created with default values (null) which is why this could happen in testing. However if changed to a nonzero value, it would require waiting delay duration before the proposal could be executed which would prevent the repayment of the flash loan in a single execution. Since parties are expected to have a nonzero execution delay, we are less concerned about the flash loan aspect of this attack.

> This finding did prompt us to consider the risk of majority attacks more broadly, where an individual can contribute and become a majority voter in a party (flash loan or not) and take control of the party. We acknowledged the majority attack before audit and don't consider it a vulnerability. Our reasoning is (1) our governance model prioritizes simplicity and speed of coordination which would be sacrificed by introducing more complex mechanisms to robustly protect against majority attacks and (2) the expectation is parties will have reasonable governance settings and active governance to veto malicious proposals to manage the risk of a majority attack and if they don't (e.g. set an execution delay of 0) it is a deliberate choice on their part rather than a vulnerability.

[H-03] Users wouldn't refund from the lost ETH crowdfunds due to the lack of ETH

Submitted by hansfriese*, also found by* 0x52 *and* evan

After the ETH crowdfunds are lost, contributors wouldn't refund their funds because the crowdfunds contract doesn't have enough ETH balance.

Proof of Concept

The core flaw is `_calculateRefundAmount()` might return more refund amount than the original contribution amount.

```
function _calculateRefundAmount(uint96 votingPower) internal
    amount = (votingPower * 1e4) / exchangeRateBps;

    // Add back fee to contribution amount if applicable.
    address payable fundingSplitRecipient_ = fundingSplitRec
    uint16 fundingSplitBps_ = fundingSplitBps;
    if (fundingSplitRecipient_ != address(0) && fundingSplit
        amount = (amount * 1e4) / (1e4 - fundingSplitBps_);
    }
}
```

When users contribute to the ETH crowdfunds, it subtracts the fee from the contribution amount.

```
File: 2023-04-party\contracts\crowdfund\ETHCrowdfundBase.sol
226:         uint16 fundingSplitBps_ = fundingSplitBps;
227:         if (fundingSplitRecipient_ != address(0) && funding
228:             uint96 feeAmount = (amount * fundingSplitBps_)
229:             amount -= feeAmount;
230:         }
```

During the calculation, it calculates `feeAmount` first which is rounded down and subtracts from the contribution amount. It means the final amount after subtracting the fee would be rounded up.

So when we calculate the original amount using `_calculateRefundAmount()`, we might get a greater value.

This shows the detailed example and POC.

1. Let's assume `fundingSplitBps = 1e3(10%), exchangeRateBps = 1e4.`
2. A user contributed `1e18 - 1` wei of ETH. After subtracting the fee, the voting power was `1e18 - 1 - (1e18 - 1) / 10 = 9 * 1e17`
3. Let's assume there are no other contributors and the crowdfund was lost.
4. When the user calls `refund()`, the refund amount will be `9 * 1e17 * 1e4 / 9000 = 1e18` in `_calculateRefundAmount()`
5. So it will try to transfer 1e18 wei of ETH from the crowdfund contract that contains 1e18 - 1 wei only. As a result, the transfer will revert and the user can't refund his funds.

```
function test_refund_reverts() public {
    InitialETHCrowdfund crowdfund = _createCrowdfund({
        initialContribution: 0,
        initialContributor: payable(address(0)),
        initialDelegate: address(0),
        minContributions: 0,
        maxContributions: type(uint96).max,
        disableContributingForExistingCard: false,
        minTotalContributions: 3 ether,
        maxTotalContributions: 5 ether,
        duration: 7 days,
        fundingSplitBps: 1000, //10% fee
        fundingSplitRecipient: payable(_randomAddress()) //r
    });
    Party party = crowdfund.party();

    uint256 ethAmount = 1 ether - 1; //contribute amount

    address member = _randomAddress();
    vm.deal(member, ethAmount);

    // Contribute
    vm.prank(member);
    crowdfund.contribute{ value: ethAmount }(member, "");
    assertEq(address(member).balance, 0);
```

```
        assertEq(address(crowdfund).balance, ethAmount); //crowc

        skip(7 days);

        assertTrue(crowdfund.getCrowdfundLifecycle() == ETHCrowc

        // Claim refund
        vm.prank(member);
        uint256 tokenId = 1;
        crowdfund.refund(tokenId); //reverts as it tried to with
    }
```

Recommended Mitigation Steps

When we subtract the fee in _processContribution(), we should calculate the final amount using `1e4 - fundingSplitBps` directly. Then there will be 2 rounds down in `_processContribution()` and `_calculateRefundAmount` and the refund amount won't be greater than the original amount.

```
    if (fundingSplitRecipient_ != address(0) && fundingSplitBps_
        amount = (amount * (1e4 - fundingSplitBps_)) / 1e4;
    }
```

0xble (Party) confirmed

[H-04] `ReraiseETHCrowdfund.sol` : Multiple scenarios how pending votes might not be claimable which is a complete loss of funds for a user

Submitted by HollaDieWaldfee*, also found by* evan *and* hansfriese

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ReraiseETHCrowdfund.sol#L256-L303

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ReraiseETHCrowdfund.sol#L333-L382

This issue is about how the `ReraiseETHCrowdfund` claim functionality can be broken.

When the claim functionality is broken this means that a user cannot claim his voting power, resulting in a complete loss of funds.

The claim functionality is not broken in any case, i.e. with any configuration of the `ReraiseETHCrowdfund` contract.

However the contract can be configured in a way - and by configured I mean specifically the `minContribution`, `maxContribution`, `minTotalContributions` and `maxTotalContributions` variables - that the claim functionality breaks.

And the configurations under which it breaks are NOT edge cases. They represent the **intended use** of the contract as discussed with the sponsor.

The fact that when the contract is used as intended it can lead to a complete loss of funds for the users makes me estimate this to be “High” severity.

Proof of Concept

We first need to understand the `claim(uint256 tokenId, address contributor)` and `claimMultiple(uint96[] memory votingPowerByCard, address contributor)` functions. They essentially make up the claim functionality as all other functions regarding claiming are just wrappers around them.

Let’s first look at the `claim(uint256 tokenId, address contributor)` function. The first part of the function is what we are interested in:

Link

```
    function claim(uint256 tokenId, address contributor) public {
        // Check crowdfund lifecycle.
        {
            CrowdfundLifecycle lc = getCrowdfundLifecycle();
            if (lc != CrowdfundLifecycle.Finalized) {
                revert WrongLifecycleError(lc);
            }
        }
```

```
        uint96 votingPower = pendingVotingPower[contributor];

        if (votingPower == 0) return;

        {
            uint96 contribution = (votingPower * 1e4) / exchangeRate
            uint96 maxContribution_ = maxContribution;
            // Check that the contribution equivalent of total pendi
            // power is not above the max contribution range. This c
            // for contributors who contributed multiple times In th
            // `claimMultiple` function should be called instead. Th
            // so parties may use the minimum and maximum contributi
            // limit the voting power of each card (e.g.  a party de
            // card = 1 vote"-like governance system where each carc
            // voting power).
            if (contribution > maxContribution_) {
                revert AboveMaximumContributionsError(contribution,
            }
        }
```

What is important is that `contribution` is calculatesd as:

```
    uint96 contribution = (votingPower * 1e4) / exchangeRateBps;
```

And then `contribution` is checked that it is `<= maxContribution` :

```
    if (contribution > maxContribution_) {
        revert AboveMaximumContributionsError(contribution, maxContr
    }
```

The explanation for why this check is necessary can be seen in the comment:

```
    // This is done
    // so parties may use the minimum and maximum contribution value
    // limit the voting power of each card (e.g.  a party desiring a
```

```
    // card = 1 vote"-like governance system where each card has equ
    // voting power).
```

The `claimMultiple(uint96[] memory votingPowerByCard, address contributor)` function allows to divide the pending voting power across multiple party cards and it employs the following checks:

Link

```
        uint96 minContribution_ = minContribution;
        uint96 maxContribution_ = maxContribution;
        for (uint256 i; i < votingPowerByCard.length; ++i) {
            if (votingPowerByCard[i] == 0) continue;

            // Check that the contribution equivalent of voting powe
            // contribution range. This is done so parties may use t
            // and maximum contribution values to limit the voting p
            // card (e.g. a party desiring a "1 card = 1 vote"-like
            // system where each card has equal voting power).
            uint96 contribution = (votingPowerByCard[i] * 1e4) / exc
            if (contribution < minContribution_) {
                revert BelowMinimumContributionsError(contribution,
            }

            if (contribution > maxContribution_) {
                revert AboveMaximumContributionsError(contribution,
            }

            votingPower -= votingPowerByCard[i];

            // Mint contributor a new party card.
            uint256 tokenId = party.mint(contributor, votingPowerByC

            emit Claimed(contributor, tokenId, votingPowerByCard[i])
        }

        // Requires that all voting power is claimed because the cor
```

```
        // expected to have burned their crowdfund NFT.
        if (votingPower != 0) revert RemainingVotingPowerAfterClaimE
    }
```

We can see that for each party card the `contribution` needs to be `>= minContribution` and `<= maxContribution`. Also the function must deal with all the voting power, so after the function call all pending voting power must be processed:

```
    if (votingPower != 0) revert RemainingVotingPowerAfterClaimError
```

Now we are in a position to look at a simple scenario how a user can end up without being able to claim his pending voting power (Note that this can also be a griefing attack whereby an attacker contributes for the victim some possibly small amount thereby making it impossible for the victim to claim):

(The test should be added to the `ReraiseETHCrowdfund.t.sol` test file)

```
    function test_cannotClaim1() public {
            ReraiseETHCrowdfund crowdfund = _createCrowdfund({
                initialContribution: 0,
                initialContributor: payable(address(0)),
                initialDelegate: address(0),
                minContributions: 0.9 ether,
                maxContributions: 1 ether,
                disableContributingForExistingCard: false,
                minTotalContributions: 1 ether,
                maxTotalContributions: 1.5 ether,
                duration: 7 days,
                fundingSplitBps: 0,
                fundingSplitRecipient: payable(address(0))
            });

            address member = _randomAddress();
            vm.deal(member, 2 ether);

            // Contribute
            vm.startPrank(member);
            crowdfund.contribute{ value: 1 ether }(member, "");
            crowdfund.contribute{ value: 1 ether }(member, "");
```

```
        vm.stopPrank();

        assertEq(crowdfund.pendingVotingPower(member), 1.5 ether

        vm.expectRevert(
            abi.encodeWithSelector(
                ETHCrowdfundBase.AboveMaximumContributionsError.
                15000000000000000000,
                10000000000000000000
            )
        );
        crowdfund.claim(member);
    }
```

In this test the following values were chosen for the important variables that I mentioned above:

```
minContribution = 0.9e18
maxContribution = 1e18

minTotalContributions = 1e18
maxTotalContributions = 1.5e18
```

What happens in the test is that first `1 ETH` is contributed then another `0.5 ETH` is contributed (It says `1 ETH` but `maxTotalContributions` is hit and so only `0.5 ETH` is contributed and the crowdfund is finalized).

The call to the `claim` function fails because `contribution = 1.5 ETH` which is above `maxContribution`.

The important thing is now to understand that `claimMultiple` can also not be called (therefore the pending voting power cannot be claimed at all).

When we call `claimMultiple` the contribution for the first party card must be in the range `[0.9e18, 1e18]` to succeed and therefore the second contribution can only be in the range of `[0.5e18,0.6e18]` which is below `minContribution` and therefore it is not possible to distribute the voting power across cards such that the call succeeds.

What we discussed so far could be mitigated by introducing some simple checks when setting up the crowdfund. The sort of checks required are like "`minTotalContributions` must be divisible by `minContribution`". I won't go into this deeply however because these checks are insufficient when we introduce a funding fee.

Let's consider a case with:

```
minContribution = 1e18
maxContribution = 1e18
minTotalContributions = 2e18
maxTotalContributions = 2e18
```

(Note that setting up the crowdfund with `minContribution==maxContribution` is an important use case where the party wants to enforce a "1 card = 1 vote"-policy).

There should be no way how this scenario causes a problem right? The contribution of a user can only be `1e18` or `2e18` and in both cases the checks in the claim functions should pass. - No

It breaks when we introduce a fee. Say there is a 1% fee (`fundingSplitBps=100`).

The contribution is calculated as (as we know from above):

(Also note that `exchangeRateBps=1e4` for all tests, i.e. the exchange rate between ETH and votes is 1:1)

```
uint96 contribution = (votingPower * 1e4) / exchangeRateBps;
```

The problem is that `votingPower` has been reduced by 1% due to the funding fee. So when a user initially contributes `1e18`, the `contribution` here is calculated to be `0.99e18 * 1e4 / 1e4 = 0.99e18` which is below `minContribution` and claiming is not possible.

Let's make a final observation: The parameters can also be such that due to rounding a similar thing happens:

```
minContribution = 1e18 + 1 Wei
maxContribution = 1e18 + 1 Wei
minTotalContributions = 2e18 + 2 Wei
maxTotalContributions = 2e18 + 2 Wei
```

Due to rounding (when calculating the funding fee or when there is not a 1:1 exchange rate) the 1 Wei in the contribution can be lost (or some other small amount) and thereby when calling `claim`, the `contribution` which has been rounded down is below `minContribution` and the claim fails.

To summarize we have seen 3 scenarios. It is not possible for me to provide an overview of all the things that can go wrong. There are just too many variables. I come back to this point in my recommendation.

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

A part of the fix is straightforward. However this is not a full fix.

I recommend to implement a functionality for claiming that cannot be blocked. I know that this may cause the “1 card = 1 vote”-policy to be violated and it may also cause `minContribution` or `maxContribution` to be violated. But maybe this is the price to pay to ensure that users can always claim.

An alternative solution may be to reduce the range of possible configurations of the crowdfund drastically such that it can be mathematically proven that users are always able to claim.

That being said there is an obvious flaw in the current code that has been confirmed by the sponsor.

The `contribution` amount that is calculated when claiming needs to add back the funding fee amount. I.e. if there was a 1% funding fee, the `contribution` amount should be `1e18` instead of `0.99e18`.

Partial fix:

```
diff --git a/contracts/crowdfund/ReraiseETHCrowdfund.sol b/contr
index 580623d..0b1ba9e 100644
--- a/contracts/crowdfund/ReraiseETHCrowdfund.sol
+++ b/contracts/crowdfund/ReraiseETHCrowdfund.sol
@@ -268,6 +268,13 @@ contract ReraiseETHCrowdfund is ETHCrowdfur

         {
             uint96 contribution = (votingPower * 1e4) / exchang
+
+            address payable fundingSplitRecipient_ = fundingSpl
+            uint16 fundingSplitBps_ = fundingSplitBps;
+            if (fundingSplitRecipient_ != address(0) && fundinc
+                contribution = (contribution * 1e4) / (1e4 - fu
+            }
+
             uint96 maxContribution_ = maxContribution;
             // Check that the contribution equivalent of total
             // power is not above the max contribution range. T
@@ -360,6 +367,13 @@ contract ReraiseETHCrowdfund is ETHCrowdfur
             // card (e.g. a party desiring a "1 card = 1 vote"-
             // system where each card has equal voting power).
             uint96 contribution = (votingPowerByCard[i] * 1e4)
+
+            address payable fundingSplitRecipient_ = fundingSpl
+            uint16 fundingSplitBps_ = fundingSplitBps;
+            if (fundingSplitRecipient_ != address(0) && fundinc
+                contribution = (contribution * 1e4) / (1e4 - fu
+            }
+
             if (contribution < minContribution_) {
                 revert BelowMinimumContributionsError(contribut
             }
```

Oxble (Party) commented:

> Additional to the partial fix recommended in the mitigation, this will be mitigated by **preventing the case where `minContribution` may be bypassed for the last contributor when the remaining contribution is less than the minimum**.

[H-05] `ReraiseETHCrowdfund.sol` : party card transfer can be front-run by claiming pending voting power which results in a loss of the voting power

Submitted by HollaDieWaldfee

In this report I show how an attacker can abuse the fact that anyone can call `ReraiseETHCrowdfund.claim` for any user and add voting power to an existing party card.

The result can be a griefing attack whereby the victim loses voting power. In some cases the attacker can take advantage himself.

In short this is what needs to happen:

1. The victim sends a transaction to transfer one of his party cards
2. The transaction is front-run and pending voting power of the victim from the `ReraiseETHCrowdfund` contract is claimed to this party card that is transferred
3. The victim thereby loses the pending voting power

The fact that any user is at risk that has pending voting power and transfers a party card and that voting power is arguably the most important asset in the protocol makes me estimate this to be “High” severity.

Proof of Concept

We start by observing that when the `ReraiseETHCrowdfund` is won, any user can call `ReraiseETHCrowdfund.claim` for any other user and either mint a new party card to him or add the pending voting power to an existing party card:

Link

```
    /// @notice Claim a party card for a contributor if the crowdfur
    ///         to claim for self or on another's behalf.
    /// @param tokenId The ID of the party card to add voting power
    ///                new card will be minted.
    /// @param contributor The contributor to claim for.
    function claim(uint256 tokenId, address contributor) public {
        // Check crowdfund lifecycle.
```

```
{
    CrowdfundLifecycle lc = getCrowdfundLifecycle();
    if (lc != CrowdfundLifecycle.Finalized) {
        revert WrongLifecycleError(lc);
    }
}


uint96 votingPower = pendingVotingPower[contributor];


if (votingPower == 0) return;


{
    uint96 contribution = (votingPower * 1e4) / exchangeRate
    uint96 maxContribution_ = maxContribution;
    // Check that the contribution equivalent of total pendi
    // power is not above the max contribution range. This c
    // for contributors who contributed multiple times In th
    // `claimMultiple` function should be called instead. Th
    // so parties may use the minimum and maximum contributi
    // limit the voting power of each card (e.g.  a party de
    // card = 1 vote"-like governance system where each carc
    // voting power).
    if (contribution > maxContribution_) {
        revert AboveMaximumContributionsError(contribution,
    }
}


// Burn the crowdfund NFT.
_burn(contributor);


delete pendingVotingPower[contributor];


if (tokenId == 0) {
    // Mint contributor a new party card.
    tokenId = party.mint(contributor, votingPower, delegatic
} else if (disableContributingForExistingCard) {
    revert ContributingForExistingCardDisabledError();
} else if (party.ownerOf(tokenId) == contributor) {
    // Increase voting power of contributor's existing party
    party.addVotingPower(tokenId, votingPower);
```

```
        } else {
            revert NotOwnerError();
        }


        emit Claimed(contributor, tokenId, votingPower);
    }
```

Note that the caller can specify any `contributor` and can add the pending votes to an existing party card if `!disableContributingForExistingCard && party.ownerOf(tokenId) == contributor`.

So if User A has pending voting power and transfers one of his party cards to User B, then User C might front-run this transfer and claim the pending voting power to the party card that is transferred.

If User B performs this attack it is not a griefing attack since User B benefits from it.

Note that at the time of sending the transfer transaction the `ReraiseETHCrowdfund` does not have to be won already. The transaction that does the front-running might contribute to the crowdfund such that it is won and then claim the pending voting power.

Add the following test to the `ReraiseETHCrowdfund.t.sol` test file. It shows how an attacker would perform such an attack:

```
    function test_FrontRunTransfer() public {
        ReraiseETHCrowdfund crowdfund = _createCrowdfund({
            initialContribution: 0,
            initialContributor: payable(address(0)),
            initialDelegate: address(0),
            minContributions: 0,
            maxContributions: type(uint96).max,
            disableContributingForExistingCard: false,
            minTotalContributions: 2 ether,
            maxTotalContributions: 3 ether,
            duration: 7 days,
            fundingSplitBps: 0,
            fundingSplitRecipient: payable(address(0))
        });
```

```
        address attacker = _randomAddress();
        address victim = _randomAddress();
        vm.deal(victim, 2.5 ether);
        vm.deal(attacker, 0.5 ether);

        // @audit-info the victim owns a party card
        vm.prank(address(party));
        party.addAuthority(address(this));
        party.increaseTotalVotingPower(1 ether);
        uint256 victimTokenId = party.mint(victim, 1 ether, address(

        vm.startPrank(victim);
        crowdfund.contribute{ value: 2.5 ether }(victim, "");
        vm.stopPrank();

        /* @audit-info
        The victim wants to transfer the party card, say to the atta
        front-runs this by completing the crowdfund and claiming the
        power to the existing party card
        */

        vm.startPrank(attacker);
        crowdfund.contribute{ value: 0.5 ether }(attacker, "");
        crowdfund.claim(victimTokenId,victim);
        vm.stopPrank();

        /* @audit-info
        when the victim's transfer is executed, he transfers also al
        that was previously his pending voting power (effectively lo
        */
        vm.prank(victim);
        party.tranferFrom(victim,attacker,victimTokenId);
    }
```

So when there is an ongoing crowdfund it is never safe to transfer one's party card. It can always result in a complete loss of the pending voting power.

Tools Used

VSCode

Recommended Mitigation Steps

In the `ReraiseETHCrowdfund.claim` function it should not be possible to add the pending voting power to an existing party card. It is possible though to allow it for the `contributor` himself but not for any user.

```
diff --git a/contracts/crowdfund/ReraiseETHCrowdfund.sol b/contr
index 580623d..cb560e1 100644
--- a/contracts/crowdfund/ReraiseETHCrowdfund.sol
+++ b/contracts/crowdfund/ReraiseETHCrowdfund.sol
@@ -292,7 +292,7 @@ contract ReraiseETHCrowdfund is ETHCrowdfunc
             tokenId = party.mint(contributor, votingPower, dele
         } else if (disableContributingForExistingCard) {
             revert ContributingForExistingCardDisabledError();
-        } else if (party.ownerOf(tokenId) == contributor) {
+        } else if (party.ownerOf(tokenId) == contributor && cor
             // Increase voting power of contributor's existing
             party.addVotingPower(tokenId, votingPower);
         } else {
```

Oxble (Party) confirmed and commented:

> Good finding, still thinking about the mitigation.

> Slightly hesitant to make the only action when claiming for someone else to be minting them a new card although minting to their existing card might be a rare action because of the friction involved in having to get the ID of one of the person's cards first. Someone minting for someone else might just find it more convenient to mint them a new card, so having that be the only action might not be much of a loss.

Oxble (Party) commented:

> We've decided to refactor the way claiming works in the `ReraiseETHCrowdfund`, partially because a large number of findings like this being submitted around that one area that highlighted for us the need to rework its logic.

> The change will make it so (1) crowdfund NFTs are minted per contribution instead of per address and (2) claiming works more like a 1:1 conversion of your

> crowdfund NFT into a party card instead of how it works now. In the future we will also add the ability to split/merge party cards.

> This should mitigate this finding because in this new system you cannot decide to add the voting power from a crowdfund NFT to an existing party card when claiming, only mint a new party card.

[H-06] `ETHCrowdfundBase.sol : totalVotingPower` is increased too much in the `_finalize` function

Submitted by **HollaDieWaldfee**, *also found by* **hansfriese**

This issue is about how the `ETHCrowdfundBase._finalize` functions calls `PartyGovernanceNFT.increaseTotalVotingPower` with an amount that does not reflect the sum of the individual users' voting power.

Thereby it will become impossible to reach unanimous votes. In other words and more generally the users' votes are worth less than they should be as the percentage is calculated against a total amount that is too big.

In short, this is how the issue is caused:

1. The voting power that a user receives is based on the amount they contribute MINUS funding fees
2. The amount of voting power by which `totalVotingPower` is increased is based on the total contributions WITHOUT subtracting funding fees

Proof of Concept

Let's first look at the affected code and then at the PoC.

The `votingPower` that a user receives for making a contribution is calculated in the `ETHCrowdfundBase._processContribution` function.

We can see that first the funding fee is subtracted and then with the lowered `amount`, the `votingPower` is calculated:

Link

```
// Subtract fee from contribution amount if applicable.
address payable fundingSplitRecipient_ = fundingSplitRecipient;
uint16 fundingSplitBps_ = fundingSplitBps;
if (fundingSplitRecipient_ != address(0) && fundingSplitBps_ > (
    uint96 feeAmount = (amount * fundingSplitBps_) / 1e4;
    amount -= feeAmount;
}


// Calculate voting power.
votingPower = (amount * exchangeRateBps) / 1e4;
```

Even before that, `totalContributions` has been increased by the full `amount` (funding fees have not been subtracted yet):

Link

```
uint96 newTotalContributions = totalContributions + amount;
uint96 maxTotalContributions_ = maxTotalContributions;
if (newTotalContributions >= maxTotalContributions_) {
    totalContributions = maxTotalContributions_;


    // Finalize the crowdfund.
    // This occurs before refunding excess contribution to act a
    // reentrancy guard.
    _finalize(maxTotalContributions_);


    // Refund excess contribution.
    uint96 refundAmount = newTotalContributions - maxTotalContri
    if (refundAmount > 0) {
        amount -= refundAmount;
        payable(msg.sender).transferEth(refundAmount);
    }
} else {
    totalContributions = newTotalContributions;
}
```

(Note that the above code looks more complicated than it is because it accounts for the fact that `maxTotalContributions` might be reached. But this is not important

for explaining this issue)

When `PartyGovernanceNFT.increaseTotalVotingPower` is called it is with the `newVotingPower` that has been calculated BEFORE funding fees are subtracted:

Link

```
uint96 newVotingPower = (totalContributions_ * exchangeRateBps)
party.increaseTotalVotingPower(newVotingPower);


// Transfer fee to recipient if applicable.
address payable fundingSplitRecipient_ = fundingSplitRecipient;
uint16 fundingSplitBps_ = fundingSplitBps;
if (fundingSplitRecipient_ != address(0) && fundingSplitBps_ > 0
    uint96 feeAmount = (totalContributions_ * fundingSplitBps_)
    totalContributions_ -= feeAmount;
    fundingSplitRecipient_.transferEth(feeAmount);
}
```

Therefore `totalVotingPower` is increased more than the sum of the voting power that the users have received.

Let's look at the PoC:

```
function test_totalVotingPower_increased_too_much() public {
        ReraiseETHCrowdfund crowdfund = _createCrowdfund({
            initialContribution: 0,
            initialContributor: payable(address(0)),
            initialDelegate: address(0),
            minContributions: 0,
            maxContributions: type(uint96).max,
            disableContributingForExistingCard: false,
            minTotalContributions: 2 ether,
            maxTotalContributions: 5 ether,
            duration: 7 days,
            fundingSplitBps: 1000,
            fundingSplitRecipient: payable(address(1))
        });

        address member1 = _randomAddress();
```

```
        address member2 = _randomAddress();
        vm.deal(member1, 1 ether);
        vm.deal(member2, 1 ether);

        // Contribute, should be allowed to update delegate
        vm.startPrank(member1);
        crowdfund.contribute{ value: 1 ether }(member1, "");
        vm.stopPrank();

        vm.startPrank(member2);
        crowdfund.contribute{ value: 1 ether }(member2, "");
        vm.stopPrank();

        skip(7 days);
        console.log(party.getGovernanceValues().totalVotingPower
        crowdfund.finalize();
        console.log(party.getGovernanceValues().totalVotingPower

        console.log(crowdfund.pendingVotingPower(member1));
        console.log(crowdfund.pendingVotingPower(member2));
    }
```

See that `totalVotingPower` is increased from `0` to `2e18`.

The voting power of both users is `0.9e18` (10% fee).

Thereby both users together receive a voting power of `1.8e18` which is only 90% of `2e18`.

Therefore it is impossible to reach an unanimous vote.

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

The fix is easy:
We must consider the funding fee when increasing the `totalVotingPower`.

Fix:

```
diff --git a/contracts/crowdfund/ETHCrowdfundBase.sol b/contract
index 4392655..3c11160 100644
--- a/contracts/crowdfund/ETHCrowdfundBase.sol
+++ b/contracts/crowdfund/ETHCrowdfundBase.sol
@@ -274,10 +274,6 @@ contract ETHCrowdfundBase is Implementatior
         // Finalize the crowdfund.
         delete expiry;

-        // Update the party's total voting power.
-        uint96 newVotingPower = (totalContributions_ * exchange
-        party.increaseTotalVotingPower(newVotingPower);
-
         // Transfer fee to recipient if applicable.
         address payable fundingSplitRecipient_ = fundingSplitRe
         uint16 fundingSplitBps_ = fundingSplitBps;
@@ -287,6 +283,10 @@ contract ETHCrowdfundBase is Implementatior
             fundingSplitRecipient_.transferEth(feeAmount);
         }

+        // Update the party's total voting power.
+        uint96 newVotingPower = (totalContributions_ * exchange
+        party.increaseTotalVotingPower(newVotingPower);
+
         // Transfer ETH to the party.
         payable(address(party)).transferEth(totalContributions_
     }
```

[Oxean (judge) increased severity to High and commented]():

> I am upgrading severity here to match #27 and will look forward to sponsor comments.

> I think this exposes a way in which there are parameter sets that leads to a loss of funds by not allowing any proposal to be executed.

[Oxble (Party) confirmed]()

[H-07] InitialETHCrowdfund + ReraiseETHCrowdfund:

`batchContributeFor` function may not refund ETH which leads to loss of funds

Submitted by HollaDieWaldfee*, also found by* evan *and* hansfriese

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/InitialETHCrowdfund.sol#L235-L268

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ReraiseETHCrowdfund.sol#L174-L202

This vulnerability exists in both the `InitialETHCrowdfund` and `ReraiseETHCrowdfund` contracts in exactly the same way.

I will continue this report by explaining the issue in only one contract. The mitigation section however contains the fix for both instances.

The `batchContributeFor` function is a wrapper that allows to make multiple calls to `contributeFor` within one function call.

It is possible to specify that this function should not revert when one individual call to `contributeFor` fails by setting `args.revertOnFailure=false`.

The issue is that in this case the ETH for a failed contribution is not refunded which leads a loss of funds for the user calling the function.

Note:
This issue also exists in the `Crowdfund.batchContributeFor` function which is out of scope. The sponsor knows about this and will fix it.

Proof of Concept

Let's look at the `batchContributeFor` function:

Link

```
function batchContributeFor(
    BatchContributeForArgs calldata args
) external payable onlyDelegateCall returns (uint96[] memory vot
    uint256 numContributions = args.recipients.length;
    votingPowers = new uint96[](numContributions);


    uint256 ethAvailable = msg.value;
    for (uint256 i; i < numContributions; ++i) {
        ethAvailable -= args.values[i];


        (bool s, bytes memory r) = address(this).call{ value: ar
            abi.encodeCall(
                this.contributeFor,
                (
                    args.tokenIds[i],
                    args.recipients[i],
                    args.initialDelegates[i],
                    args.gateDatas[i]
                )
            )
        );


        if (!s) {
            if (args.revertOnFailure) {
                r.rawRevert();
            }
        } else {
            votingPowers[i] = abi.decode(r, (uint96));
        }
    }


    // Refund any unused ETH.
    if (ethAvailable > 0) payable(msg.sender).transfer(ethAvaila
}
```

We can see that `ethAvailable` is reduced before every call to `contributeFor`:

```
ethAvailable -= args.values[i];
```

But it is only checked later if the call was successful:

```
    if (!s) {
        if (args.revertOnFailure) {
            r.rawRevert();
        }
```

And if `args.revertOnFailure=false` there is no revert and `ethAvailable` is not increased again.

Therefore the user has to pay for failed contributions.

Add the following test to the `InitialETHCrowdfund.t.sol` test file:

```
    function test_batchContributeFor_noETHRefund() public {
        InitialETHCrowdfund crowdfund = _createCrowdfund({
            initialContribution: 0,
            initialContributor: payable(address(0)),
            initialDelegate: address(0),
            minContributions: 1 ether,
            maxContributions: type(uint96).max,
            disableContributingForExistingCard: false,
            minTotalContributions: 3 ether,
            maxTotalContributions: 5 ether,
            duration: 7 days,
            fundingSplitBps: 0,
            fundingSplitRecipient: payable(address(0))
        });
        Party party = crowdfund.party();

        address sender = _randomAddress();
        vm.deal(sender, 2.5 ether);

        // Batch contribute for
        vm.prank(sender);
        uint256[] memory tokenIds = new uint256[](3);
        address payable[] memory recipients = new address payable[](
        address[] memory delegates = new address[](3);
        uint96[] memory values = new uint96[](3);
        bytes[] memory gateDatas = new bytes[](3);
        for (uint256 i; i < 3; ++i) {
```

```
            recipients[i] = _randomAddress();
            delegates[i] = _randomAddress();
            values[i] = 1 ether;
        }

        // @audit-info set values[2] = 0.5 ether such that contribut
        values[2] = 0.5 ether;

        uint96[] memory votingPowers = crowdfund.batchContributeFor{
            InitialETHCrowdfund.BatchContributeForArgs({
                tokenIds: tokenIds,
                recipients: recipients,
                initialDelegates: delegates,
                values: values,
                gateDatas: gateDatas,
                revertOnFailure: false
            })
        );

        // @audit-info balance of sender is 0 ETH even though 0.5 ET
        assertEq(address(sender).balance, 0 ether);
    }
```

The `sender` sends 2.5 ETH and 1 of the 3 contributions fails since `minContribution` is above the amount the `sender` wants to contribute (Note that in practice there are more ways for the contribution to fail).

The sender's balance in the end is 0 ETH which shows that there is no refund.

Tools Used

VSCode,Foundry

Recommended Mitigation Steps

The following changes need to be made to the `InitialETHCrowdfund` and `ReraiseETHCrowdfund` contracts:

```
diff --git a/contracts/crowdfund/InitialETHCrowdfund.sol b/contr
index 8ab3b5c..19e09ac 100644
--- a/contracts/crowdfund/InitialETHCrowdfund.sol
+++ b/contracts/crowdfund/InitialETHCrowdfund.sol
```

```
@@ -240,8 +240,6 @@ contract InitialETHCrowdfund is ETHCrowdfunc

         uint256 ethAvailable = msg.value;
         for (uint256 i; i < numContributions; ++i) {
-            ethAvailable -= args.values[i];
-
             (bool s, bytes memory r) = address(this).call{ valu
                 abi.encodeCall(
                     this.contributeFor,
@@ -260,6 +258,7 @@ contract InitialETHCrowdfund is ETHCrowdfunc
                 }
             } else {
                 votingPowers[i] = abi.decode(r, (uint96));
+                ethAvailable -= args.values[i];
             }
         }
```

```
diff --git a/contracts/crowdfund/ReraiseETHCrowdfund.sol b/contr
index 580623d..ad70b27 100644
--- a/contracts/crowdfund/ReraiseETHCrowdfund.sol
+++ b/contracts/crowdfund/ReraiseETHCrowdfund.sol
@@ -179,8 +179,6 @@ contract ReraiseETHCrowdfund is ETHCrowdfunc

         uint256 ethAvailable = msg.value;
         for (uint256 i; i < numContributions; ++i) {
-            ethAvailable -= args.values[i];
-
             (bool s, bytes memory r) = address(this).call{ valu
                 abi.encodeCall(
                     this.contributeFor,
@@ -194,6 +192,7 @@ contract ReraiseETHCrowdfund is ETHCrowdfunc
                 }
             } else {
                 votingPowers[i] = abi.decode(r, (uint96));
+                ethAvailable -= args.values[i];
             }
         }
```

Now `ethAvailable` is only reduced when the call to `contributeFor` was successful.

Oxean (judge) commented:

> Would welcome comment on this issue. AFAICT, this leads to a direct loss of user funds, which makes me think that a High severity is warranted. There is no external pre-condition(s) required for this to happen.

HollaDieWaldfee (warden) commented:

> @0xean - Yeah you are right, it leads to a direct loss of funds and there are no preconditions. Should have set it to “High” probably.

0xble (Party) confirmed

0xean (judge) increased severity to High

[H-08] `VetoProposal` : User can veto multiple times so every proposal can be vetoed by any user that has a small amount of votes

Submitted by **HollaDieWaldfee**, *also found by* **0x52** *and* **hansfriese**

The `VetoProposal` contract allows to veto proposals with the `voteToVeto` function.

When the amount of votes collected to veto a proposal exceeds a certain threshold (the `passThresholdBps` , which is determined upon initialization of the party), the proposal is vetoed, meaning it cannot execute anymore (its status becomes `Defeated`).

The `passThresholdBps` specifies a percentage of the `totalVotingPower` of the party.

E.g. `passThresholdBps=1000` means that 10% of the `totalVotingPower` must veto a proposal such that the veto goes through.

The issue is that the contract lacks the obvious check that a user has not vetoed before, thereby a user can veto multiple times.

So say a user holds 1% of `totalVotingPower` and in order for the veto to go through, 10% of `totalVotingPower` must veto.

The user can just veto 10 times to reach the 10% requirement.

The impact is obvious: Any user with a small amount of votes can veto any proposal. This is a critical bug since the party may become unable to perform any actions if there is a user that vetoes all proposals.

Proof of Concept

Add the following test to the `VetoProposal.t.sol` test file:

```
    function test_VetoMoreThanOnce() public {
        _assertProposalStatus(PartyGovernance.ProposalStatus.Voting)

        // Vote to veto
        vm.prank(voter1);
        vetoProposal.voteToVeto(party, proposalId, 0);

        _assertProposalStatus(PartyGovernance.ProposalStatus.Voting)
        assertEq(vetoProposal.vetoVotes(party, proposalId), 1e18);

        // Vote to veto (passes threshold)
        vm.prank(voter1);
        vetoProposal.voteToVeto(party, proposalId, 0);

        _assertProposalStatus(PartyGovernance.ProposalStatus.Defeate
        assertEq(vetoProposal.vetoVotes(party, proposalId), 0); // C
    }
```

In the test file, these are the conditions: `totalVotingPower = 3e18`, required votes threshold is 51%, `voter1` has `1e18` votes which is `~33%`. Clearly `voter1` should not be able to veto the proposal on his own.

You can see in the test that `voter1` can veto 2 times.

After the first call to `voteToVeto`, the threshold is not yet reached (the proposal is still in the `Voting` state).

After the second call to `voteToVeto` the threshold is reached and the proposal is in the `Defeated` state.

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

The fix is straightforward.

We introduce a `hasVoted` mapping that tracks for each `(party, proposalId, address)` triplet if it has vetoed already.

Fix:

```
diff --git a/contracts/proposals/VetoProposal.sol b/contracts/pr
index 780826f..fb1f1ab 100644
--- a/contracts/proposals/VetoProposal.sol
+++ b/contracts/proposals/VetoProposal.sol
@@ -8,9 +8,11 @@ import "../party/Party.sol";
 contract VetoProposal {
     error NotPartyHostError();
     error ProposalNotActiveError(uint256 proposalId);
+    error AlreadyVotedError(address caller);

     /// @notice Mapping from party to proposal ID to votes to v
     mapping(Party => mapping(uint256 => uint96)) public vetoVot
+    mapping(Party => mapping(uint256 => mapping(address => bool

     /// @notice Vote to veto a proposal.
     /// @param party The party to vote on.
@@ -33,6 +35,12 @@ contract VetoProposal {
         if (proposalStatus != PartyGovernance.ProposalStatus.Vo
             revert ProposalNotActiveError(proposalId);

+        if (hasVoted[party][proposalId][msg.sender]) {
+            revert AlreadyVotedError(msg.sender);
+        }
+
+        hasVoted[party][proposalId][msg.sender] = true;
+
```

```
        // Increase the veto vote count
        uint96 votingPower = party.getVotingPowerAt(
            msg.sender,
```

0xble (Party) confirmed

Medium Risk Findings (12)

[M-01] Use of `_mint` in `ReraiseETHCrowdfund#_contribute` is incompatible with `PartyGovernanceNFT#mint`

Submitted by 0x52

Misconfigured receiver could accidentally DOS party.

Proof of Concept

ReraiseETHCrowdfund.sol#L238

```
        if (previousVotingPower == 0) _mint(contributor); <- @audit-
```

ReraiseETHCrowdfund.sol#L374

```
            uint256 tokenId = party.mint(contributor, votingPowerBy(
```

PartyGovernanceNFT.sol#L162

```
        _safeMint(owner, tokenId); <- @audit-issue PartyGovernanceNF
```

The issue at hand is that ReraiseETHCrowdfund#_contribute and PartyGovernanceNFT#mint use inconsistent minting methods. PartyGovernanceNFT uses safeMint whereas ReraiseETHCrowdfund uses the standard mint. This is problematic because this means that a contract that doesn't implement

ERC721Receiver can receive a CrowdfundNFT but they can never claim because safeMint will always revert. This can cause a party to be inadvertently DOS'd because CrowdfundNFTs are soul bound and can't be transferred.

Recommended Mitigation Steps

Use _safeMint instead of _mint for ReraiseETHCrowdfund#_contribute

0xble (Party) confirmed and commented:

> I think a better mitigation would be to allow the user to specify a `receiver` address that can receive the party NFT when claiming, so if they cannot claim themselves they can specify another address that should receive it instead. It works similarly in `Crowdfund`, used to implement prior crowdfunds.

[M-02] `MaxContribution` check can be bypassed to give a card high voting power

Submitted by **evan**, *also found by* **hansfriese**

ReraiseETHCrowdfund tries limit the voting power of each card by doing a min/maxContribution check in claim and claimMultiple.

```
uint96 contribution = (votingPower * 1e4) / exchange
uint96 maxContribution_ = maxContribution;
// Check that the contribution equivalent of total p
// power is not above the max contribution range. Th
// for contributors who contributed multiple times l
// `claimMultiple` function should be called instead
// so parties may use the minimum and maximum contri
// limit the voting power of each card (e.g.  a part
// card = 1 vote"-like governance system where each
// voting power).
if (contribution > maxContribution_) {
    revert AboveMaximumContributionsError(contributi
}
```

https://github.com/code-423n4/2023-04-party/blob/main/contracts/crowdfund/ReraiseETHCrowdfund.sol#L270-L282

```
            // Check that the contribution equivalent of voting
            // contribution range. This is done so parties may u
            // and maximum contribution values to limit the voti
            // card (e.g. a party desiring a "1 card = 1 vote"-l
            // system where each card has equal voting power).
            uint96 contribution = (votingPowerByCard[i] * 1e4) /
            if (contribution < minContribution_) {
                revert BelowMinimumContributionsError(contributi
            }

            if (contribution > maxContribution_) {
                revert AboveMaximumContributionsError(contributi
            }
```

https://github.com/code-423n4/2023-04-party/blob/main/contracts/crowdfund/ReraiseETHCrowdfund.sol#L357-L369

However, this check can be bypassed due to the following code segment

```
        else if (party.ownerOf(tokenId) == contributor) {
            // Increase voting power of contributor's existing p
            party.addVotingPower(tokenId, votingPower);
        }
```

https://github.com/code-423n4/2023-04-party/blob/main/contracts/crowdfund/ReraiseETHCrowdfund.sol#L295-L298

Consider the following situation. Suppose ReraiseETHCrowdfund sets maximumContribution to only allow at most 3 units of voting power in each card. Some user X can contribute the maximum amount twice as 2 different contributor addresses A & B (both of which he controls). When the crowdfund has finalized, X can first call claim as A, then transfer the partyGovernanceNFT from A to B (note that while the crowdfundNFT can't be transferred, the partyGovernanceNFT can be transferred), and finally call claim as B to get a card with 6 units of voting power.

Impact

The degree of impact really depends on the use case of the party. Some parties would like each card to represent a single vote - this would obviously violate that. Generally, it's not a great idea to allow a single card to hold a high amount of votes, so I'll leave this as a medium for now.

Recommended Mitigation Steps

One solution is to restrict the maximum voting power on partyGovernanceNFT's side. It can check the votingPower of each card before adding more votingPower to it.

0xble (Party) commented:

> @evan - Couldn't this also be mitigated by checking the voting power held by a card before adding voting power to it?

0xble (Party) confirmed and commented:

> Considering a refactor to only check `contribution > maxContribution` in `claim()` if `disableContributingForExistingCard` is true, which would mitigate this.

> The reasoning is the check was added for the case to support parties that wish to have a "1 card, 1 vote"-type governance system and must have fixed voting power per card. To do this we would expect `disableContributingForExistingCard` to be enabled. If it is not, the creator would be indicating it doesn't matter.

0xble (Party) commented:

> We've decided to refactor the way claiming works in the `ReraiseETHCrowdfund`, partially because a large number of findings like this being submitted around that one area that highlighted for us the need to rework its logic.

> The change will make it so (1) crowdfund NFTs are minted per contribution instead of per address and (2) claiming works more like a 1:1 conversion of your crowdfund NFT into a party card instead of how it works now. In the future we will also add the ability to split/merge party cards.

> This should mitigate this finding because the voting power transferred from crowdfund NFT to party card is known to be within the min/max contribution limit (otherwise the contribution that created the crowdfund NFT would have reverted) so there is no longer a min/max contribution check required when claiming.

[M-03] Contributions can be smaller than `minContribution` and may receive no voting power

Submitted by Ox52

Valid contribution is awarded no voting power.

Proof of Concept

ETHCrowdfundBase.sol#L195-L219

```
        uint96 minContribution_ = minContribution;
        if (amount < minContribution_) {
            revert BelowMinimumContributionsError(amount, minContrib
        }
        uint96 maxContribution_ = maxContribution;
        if (amount > maxContribution_) {
            revert AboveMaximumContributionsError(amount, maxContrib
        }

        uint96 newTotalContributions = totalContributions + amount;
        uint96 maxTotalContributions_ = maxTotalContributions;
        if (newTotalContributions >= maxTotalContributions_) {
            totalContributions = maxTotalContributions_;

            // Finalize the crowdfund.
            // This occurs before refunding excess contribution to a
            // reentrancy guard.
            _finalize(maxTotalContributions_);

            // Refund excess contribution.
            uint96 refundAmount = newTotalContributions - maxTotalCo
            if (refundAmount > 0) {
                amount -= refundAmount; <- @audit-issue amount is re
                payable(msg.sender).transferEth(refundAmount);
            }
```

When processing a contribution, if the amount contributed would push the crowdfund over the max then it is reduced. This is problematic because this reduction occurs AFTER it checks the amount against the minimum contribution. The result is that these contributions can end up being less than the specified minimum.

Although an edge case, if amount is smaller than exchangeRateBps as it could result in the user receiving no voting power at all for their contribution.

Recommended Mitigation Steps

Enforce minContribution after reductions to amount.

Oxble (Party) acknowledged and commented:

> This was done intentionally. `minContribution` may be bypassed for the last contributor when the remaining contribution is less than the min otherwise the party may never be able to reach `maxTotalContributions`.

> "Although an edge case, if amount is smaller than exchangeRateBps as it could result in the user receiving no voting power at all for their contribution."

> For this though we can add a check to ensure that `votingPower != 0`.

Oxble (Party) confirmed and commented:

> Thinking about this one more, it may be worth enforcing the min/max contribution checks after this happens to not allow for this case where `contribution < minContribution`. The way it currently is creates a potentially bad scenario, as pointed out, where the last contributor may receive no voting power or not be able to claim their party card if this was a `ReraiseETHCrowdfund`.

> The motivation for allowing `minContribution` to be bypassed for the last contributor when the remaining contribution is less than `minContribution` was to allow parties to reach `maxTotalContributions` in a case where they otherwise wouldn't be able to because the contribution to reach it would be below `minContribution`. However, in this scenario the crowdfund can still be won if either the crowdfund expires above `minTotalContribution` or a host finalizes at

> any point after `minTotalContribution` has been reached. So given the party has recourse if this were to happen, it makes more sense not to allow for this edge case where `contribution < minContribution`.

> Will mitigate by enforcing `minContribution` after reductions to the contribution amount.

[M-04] `ReraiseETHCrowdfund#claimMultiple` can be used to grief large depositors

Submitted by 0x52

User can be grieved by being force minted a large number of NFTs with low voting power instead of one with high voting power.

Proof of Concept

ReraiseETHCrowdfund.sol#L354-L377

```
        for (uint256 i; i < votingPowerByCard.length; ++i) {
            if (votingPowerByCard[i] == 0) continue;

            uint96 contribution = (votingPowerByCard[i] * 1e4) / exc
            if (contribution < minContribution_) {
                revert BelowMinimumContributionsError(contribution,
            }

            if (contribution > maxContribution_) {
                revert AboveMaximumContributionsError(contribution,
            }

            votingPower -= votingPowerByCard[i];

            // Mint contributor a new party card.
            uint256 tokenId = party.mint(contributor, votingPowerBy(

            emit Claimed(contributor, tokenId, votingPowerByCard[i])
        }
```

`ReraiseETHCrowdfund#claimMultiple` can be called by any user for any other user. The above loop uses the user specified `votingPowerByCard` to assign each token a voting power and mint them to the contributor. This is problematic because large contributors can have their voting power fragmented into a large number of NFTs which a small amount of voting power each. The dramatically inflates the gas costs of the affected user.

Example:

`minContribution = 1` and `maxContribution = 100`. User A contributes 100. This means they should qualify for one NFT of the largest size. However instead they can be minted 100 NFTs with 1 vote each.

Recommended Mitigation Steps

If msg.sender isn't contributor it should force the user to mint the minimum possible number of NFTs:

```
        uint256 votingPower = pendingVotingPower[contributor];

        if (votingPower == 0) return;

+   if (msg.sender != contributor) {
+       require(votingPowerByCard.length == (((votingPower - 1)/n
+   }
```

Oxean (judge) commented:

> Looking forward to sponsor comment on this one, I do see the potential issue.

Oxble (Party) confirmed and commented:

> To give more context, the reason we allow minting/claiming on another's behalf is to allow others to potentially unblock governance if a user with delegated voting power does not come to claim, so it is an important feature to enable. It also works this way in prior releases of the protocol with past crowdfunds.

> This is a valid concern though and I like the recommended mitigation.

Oxble (Party) commented:

> We've decided to refactor the way claiming works in the `ReraiseETHCrowdfund`, partially because a large number of findings like this being submitted around that one area that highlighted for us the need to rework its logic.

> The change will make it so (1) crowdfund NFTs are minted per contribution instead of per address and (2) claiming works more like a 1:1 conversion of your crowdfund NFT into a party card instead of how it works now. In the future we will also add the ability to split/merge party cards.

> This should mitigate this finding because in this new system you cannot decide how to allocate another person's voting power when claiming for them, there is only one choice which is to convert their crowdfund NFT into a party card of equivalent voting power.

[M-05] Possible DOS attack using dust in `ReraiseETHCrowdfund._contribute()`

Submitted by hansfriese

Normal contributors wouldn't contribute to the crowdfund properly by a malicious frontrunner.

Proof of Concept

When users contribute to the `ReraiseETHCrowdfund`, it mints the crowdfund NFT in `_contribute()`.

```
File: 2023-04-party\contracts\crowdfund\ReraiseETHCrowdfund.sol
228:         votingPower = _processContribution(contributor, del
229:
230:         // OK to contribute with zero just to update delega
231:         if (amount == 0) return 0;
232:
233:         uint256 previousVotingPower = pendingVotingPower[co
234:
235:         pendingVotingPower[contributor] += votingPower;
```

```
236:
237:         // Mint a crowdfund NFT if this is their first cont
238:         if (previousVotingPower == 0) _mint(contributor); /
```

As we can see, it mints the NFT when `previousVotingPower == 0` to mint for the first contribution.

But `votingPower` from `_processContribution()` might be 0 even if `amount > 0` and `pendingVotingPower[contributor]` would be remained as 0 after the first contribution.

Then this function will revert from the second contribution as it tries to mint the NFT again.

The below shows the detailed scenario and POC.

1. Let's assume `exchangeRateBps = 5e3`. So votingPower for 1 wei is zero. Also, from the test configurations, it's not a strong condition to assume `minContributions = 0`.
2. After noticing an honest user contributes with 1 ether, an attacker frontruns `contributeFor()` for the honest user with 1 wei.
3. Then the crowdfund NFT of the honest user will be minted but the voting power is still 0.
4. During the honest user's `contribute()`, it will try to mint the NFT again as previousVotingPower == 0 and revert. So he can't contribute for this crowdfund.

While executing the POC, opts.exchangeRateBps should be `5e3`.

```
    function test_contribute_DOSByFrontrunnerWithDust() public {
        ReraiseETHCrowdfund crowdfund = _createCrowdfund({
            initialContribution: 0,
            initialContributor: payable(address(0)),
            initialDelegate: address(0),
            minContributions: 0,
            maxContributions: type(uint96).max,
            disableContributingForExistingCard: false,
            minTotalContributions: 3 ether,
```

```
            maxTotalContributions: 5 ether,
            duration: 7 days,
            fundingSplitBps: 0,
            fundingSplitRecipient: payable(address(0))
        });

        address attacker = _randomAddress();
        address honest = _randomAddress();
        vm.deal(attacker, 1); //attacker has 1 wei
        vm.deal(honest, 1 ether); //honest user has 1 ether

        // Contribute
        vm.startPrank(attacker); //attacker frontruns for the hc
        crowdfund.contributeFor{ value: 1 }(payable(honest), hor
        vm.stopPrank();

        assertEq(crowdfund.balanceOf(honest), 1); //crowdfund NF
        assertEq(crowdfund.pendingVotingPower(honest), 0); //vot

        vm.expectRevert(
            abi.encodeWithSelector(
                CrowdfundNFT.AlreadyMintedError.selector,
                honest,
                uint256(uint160(honest))
            )
        );
        vm.startPrank(honest); //when the honest user contribute
        crowdfund.contribute{ value: 1 ether }(honest, "");
        vm.stopPrank();
    }
```

Recommended Mitigation Steps

Recommend minting the crowdfund NFT when the new `votingPower` is positive. Then we can avoid duplicate mints.

```
File: 2023-04-party\contracts\crowdfund\ReraiseETHCrowdfund.sol
233:         uint256 previousVotingPower = pendingVotingPower[cc
234:
235:         pendingVotingPower[contributor] += votingPower;
236:
237:         // Mint a crowdfund NFT if this is their first mear
238:         if (previousVotingPower == 0 && votingPower != 0) _
```

Oxble (Party) confirmed and commented:

> Will be mitigated by reverting if contributing leads to zero voting power (i.e. `contributionAmount * exchangeRateBps / 1e4 == 0`).

[M-06] `PartyGovernanceNFT.sol` : burn function does not reduce `totalVotingPower` making it impossible to reach unanimous votes

Submitted by HollaDieWaldfee

With the new version of the Party protocol the `PartyGovernanceNFT.burn` function has been introduced.

This function is used to burn party cards.

According to the sponsor the initial purpose of this function was to enable the `InitialETHCrowdfund` contract (the `burn` function is needed for refunds).

Later on they decided to allow any user to call this function and to burn their party cards.

The second use case when a regular user burns his party card is when the issue occurs.

The `PartyGovernanceNFT.burn` function does not decrease `totalVotingPower` which makes it impossible to reach an unanimous vote after a call to this function and it makes remaining votes of existing users less valuable than they should be.

Proof of Concept

Let's look at the `PartyGovernanceNFT.burn` function:

Link

```
function burn(uint256 tokenId) external onlyDelegateCall {
    address owner = ownerOf(tokenId);
```

```
        if (
            msg.sender != owner &&
            getApproved[tokenId] != msg.sender &&
            !isApprovedForAll[owner][msg.sender]
        ) {
            // Allow minter to burn cards if the total voting power
            // been set (e.g. for initial crowdfunds) meaning the pa
            // yet started.
            uint96 totalVotingPower = _governanceValues.totalVotingP
            if (totalVotingPower != 0 || !isAuthority[msg.sender]) {
                revert UnauthorizedToBurnError();
            }
        }


        uint96 votingPower = votingPowerByTokenId[tokenId].safeCastU
        mintedVotingPower -= votingPower;
        delete votingPowerByTokenId[tokenId];


        _adjustVotingPower(owner, -votingPower.safeCastUint96ToInt19


        _burn(tokenId);
    }
```

It burns the party card specified by the `tokenId` parameter and makes the appropriate changes to the voting power of the owner (by calling `_adjustVotingPower`) and to `mintedVotingPower`.

But it does not reduce `totalVotingPower` which remains untouched by this function.

In case this function is called by `InitialETHCrowdfund` it is intended that `totalVotingPower` is not reduced. In this case the `burn` function is only called when the initial crowdfund is lost and `totalVotingPower` hasn't even been increased so it is still `0` (the initial value).

But why is it an issue when a regular user calls this function?

Let's consider the following scenario:

```
Alice: 100 Votes
Bob: 100 Votes
Chris: 100 Votes

totalVotingPower = 300 Votes
```

Now Alice decides to burn half of her voting power:

```
Alice: 50 Votes
Bob: 100 Votes
Chris: 100 Votes

totalVotingPower = 300 Votes
```

Now it is easy to see why it is a problem that `totalVotingPower` is not reduced.

It is impossible to reach an unanimous vote because even if all users vote there is only a `(250/300) = ~83%` agreement.

One vote only represents `1/300 = ~ 0.33%` of all votes even though it should represent `1/250 = 0.4%` of all votes. And thereby votes are less valuable than they should be.

You can see in the following test that `totalVotingPower` stays unaffected even though `voter1` burns his party card which represents a third of all votes.

(Add the test to the `PartyGovernanceNFTUnit.sol` test file and add this import: `import "../../contracts/party/PartyGovernance.sol";` to access the `GovernanceValues` struct).

```
function test_canntReachUnanimousVoteAfterBurning() external {
    _initGovernance();
    address voter1 = _randomAddress();
    address voter2 = _randomAddress();
    address voter3 = _randomAddress();
    uint256 vp = defaultGovernanceOpts.totalVotingPower / 3;
    uint256 token1 = nft.mint(voter1, vp, voter1);
```

```
        uint256 token2 = nft.mint(voter2, vp, voter2);
        uint256 token3 = nft.mint(voter3, vp, voter3);

        assertEq(nft.mintedVotingPower(), vp*3);
        assertEq(nft.getCurrentVotingPower(voter1), vp);

        PartyGovernance.GovernanceValues memory gv = nft.getGovernar
        console.log(gv.totalVotingPower);

        vm.prank(voter1);
        nft.burn(token1);
        gv = nft.getGovernanceValues();
        // totalVotingPower stays the same
        console.log(gv.totalVotingPower);
    }
```

The remaining two voters will not be able to reach unanimous vote since the `_isUnanimousVotes` function is called with `totalVotingPower` as the total votes with which to calculate the percentage.

Link

```
    if (_isUnanimousVotes(pv.votes, _governanceValues.totalVotingPow
```

Link

```
    if (_isUnanimousVotes(pv.votes, gv.totalVotingPower)) {
```

Tools Used

VSCode

Recommended Mitigation Steps

It is important to understand that when `InitialETHCrowdfund` calls the `burn` function it is intended that `totalVotingPower` is not reduced.

So we need to differentiate these two cases.

Fix:

```
diff --git a/contracts/party/PartyGovernanceNFT.sol b/contracts/
index 9ccfa1f..d382d0e 100644
--- a/contracts/party/PartyGovernanceNFT.sol
+++ b/contracts/party/PartyGovernanceNFT.sol
@@ -200,6 +200,7 @@ contract PartyGovernanceNFT is PartyGovernar
     /// @param tokenId The ID of the NFT to burn.
     function burn(uint256 tokenId) external onlyDelegateCall {
         address owner = ownerOf(tokenId);
+        uint96 totalVotingPower = _governanceValues.totalVoting
         if (
             msg.sender != owner &&
             getApproved[tokenId] != msg.sender &&
@@ -208,7 +209,6 @@ contract PartyGovernanceNFT is PartyGovernar
             // Allow minter to burn cards if the total voting p
             // been set (e.g. for initial crowdfunds) meaning t
             // yet started.
-            uint96 totalVotingPower = _governanceValues.totalVo
             if (totalVotingPower != 0 || !isAuthority[msg.sende
                 revert UnauthorizedToBurnError();
             }
@@ -218,6 +218,10 @@ contract PartyGovernanceNFT is PartyGoverna
         mintedVotingPower -= votingPower;
         delete votingPowerByTokenId[tokenId];

+        if (totalVotingPower != 0 || !isAuthority[msg.sender])
+            _governanceValues.totalVotingPower = totalVotingPow
+        }
+
         _adjustVotingPower(owner, -votingPower.safeCastUint96To

         _burn(tokenId);
```

Also note that the `|| !isAuthority[msg.sender]` part of the condition is important.

It ensures that if we are not yet in the governance phase, i.e. `totalVotingPower == 0` and a user calls the `burn` function he cannot burn his party card. This is because the `totalVotingPower - votingPower` subtraction results in an underflow.

This ensures that in the pre-governance phase a user cannot accidentally burn his party card. He can only burn it via the `InitialETHCrowdfund` contract which ensures the user gets his ETH refund.

0xble (Party) commented:

> @HollaDieWaldfee - I like the recommended mitigation. Had a question, if there were snapshots of `totalVotingPower` for each proposal is it fine that those are not updated? It is the same as if that an inactive user did not vote on a proposal, which can happen already. My concerns with updating it is that it seems like a riskier mitigation to implement that may cause unintended side-effects (if not now, later down the line). I also don't want to add more storage to the contract to fix this (it is already at with the contract size limit with `--via-ir` enabled).

HollaDieWaldfee (warden) commented:

> @0xble - Yes I think it is ok to not update the snapshots. I have the same reasoning as you that it's basically just an inactive user.

> And I agree that updating instead is wrong because it gives remaining users higher voting power.

0xble (Party) confirmed

[M-07] `totalVotingPower` needs to be snapshotted for each proposal because it can change and thereby affect consensus when accepting / vetoing proposals

Submitted by HollaDieWaldfee

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/party/PartyGovernance.sol#L598-L605

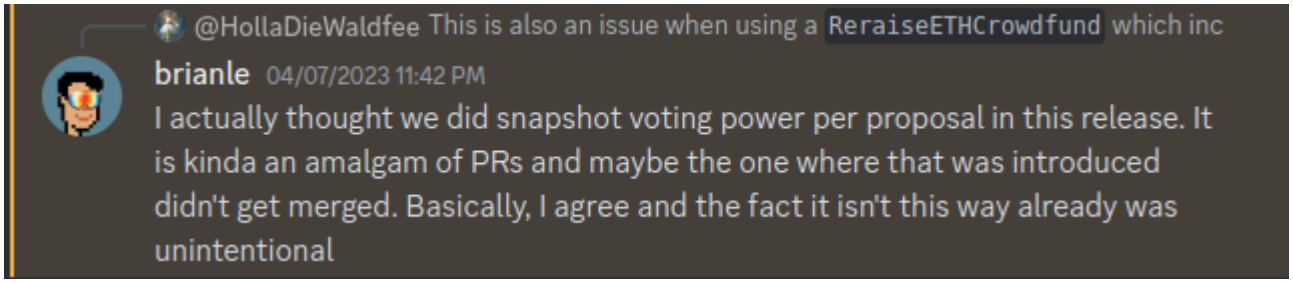
https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/proposals/VetoProposal.sol#L46-L51

This issue does not manifest itself in a limited segment of the code.

Instead it spans multiple contracts and derives its impact from the interaction of these contracts.

In the PoC section I will do my best in explaining how this results in an issue.

I discussed this with the sponsor and they explained to me that this issue is due to a PR that has unintentionally not been merged.



So they have already written the code that is necessary to fix this issue. It's just not been merged with this branch. So since the sponsor knows about this already and it's just the PR that has gone missing it's not necessary for me to provide the full Solidity code to fix this issue.

In short, this issue is due to the fact that the `totalVotingPower` is not snapshotted when a proposal is created.

The votes that are used to vote for a proposal (or veto it) are based on a specific snapshot (1 block prior to the proposal being created).

When the `totalVotingPower` changes this leads to unintended consequences.

When `totalVotingPower` decreases, votes become more valuable than they should be.

And when `totalVotingPower` increases, votes become less valuable than they should be.

Proof of Concept

When a proposal is created via the `PartyGovernance.propose` function, the proposal's `proposedTime` is set:

Link

```
    ProposalStateValues({
        proposedTime: uint40(block.timestamp),
        passedTime: 0,
        executedTime: 0,
        completedTime: 0,
        votes: 0
    }),
```

When users then vote in order to accept the proposal or veto the proposal, their votes are based on the snapshot at the `proposedTime - 1` timestamp.

We can see this in the `PartyGovernance.accept` function:

Link

```
    uint96 votingPower = getVotingPowerAt(msg.sender, values.propose
```

And we can see it in the `VetoProposal.voteToVeto` function:

Link

```
    uint96 votingPower = party.getVotingPowerAt(
        msg.sender,
        proposalValues.proposedTime - 1,
        snapIndex
    );
```

However the `totalVotingPower` to determine whether enough votes have been collected is the current `totalVotingPower`:

Link

```
    if (
        values.passedTime == 0 &&
```

```
        _areVotesPassing(
            values.votes,
            _governanceValues.totalVotingPower,
            _governanceValues.passThresholdBps
        )
    ) {
```

[Link]

```
    if (
        _areVotesPassing(
            newVotes,
            governanceValues.totalVotingPower,
            governanceValues.passThresholdBps
        )
```

The `totalVotingPower` is not constant. It can increase and decrease.

Now we can understand the issue. The `totalVotingPower` must be based on the same time as the votes (i.e. `proposedTime - 1`).

Let's look at a scenario:

```
At the time of proposal creation (proposedTime - 1):

Alice: 100 Votes
Bob: 50 Votes
Chris: 50 Votes

totalVotingPower=200
```

Let's say 80% of votes are necessary for the proposal to pass.

Now the `totalVotingPower` is increased (e.g. by a `ReraiseETHCrowdfund`) since David now has 100 Votes:

```
Alice: 100 Votes
```

```
Bob: 50 Votes
Chris: 50 Votes
David: 100 Votes

totalVotingPower=300
```

Now it is impossible for the proposal to pass.

The proposal needs 80% of 300 Votes which is 240 Votes. But the votes are used from the old snapshot and there were only 200 Votes.

The old `totalVotingPower` should have been used (200 Votes instead of 300 Votes).

Similarly there is an issue when `totalVotingPower` decreases:

```
Alice: 100 Votes
Bob: 50 Votes
Chris: 0 Votes

totalVotingPower=150
```

If 60% of the votes are necessary for the proposal to pass, Alice can make the proposal pass on her own because `totalVotingPower=150` is used even though the old `totalVotingPower=200` should be used.

Tools Used

VSCode

Recommended Mitigation Steps

As explained above the sponsor already has the code to implement snapshotting the `totalVotingPower`.

In short the following changes need to be made:

1. Snapshot `totalVotingPower` whenever it is changed

2. Whenever `totalVotingPower` is used to calculate whether a proposal is accepted / vetoed, the snapshot should be used

0xble (Party) confirmed

[M-08] `ETHCrowdfundBase.sol` : All funds are lost when fee recipient cannot receive ETH

Submitted by **HollaDieWaldfee**

In the `ETHCrowdfundBase` contract a `fundingSplitRecipient` address is configured which receives a percentage of the funds in case the crowdfund is won.

Neither the `fundingSplitRecipient` address nor the `fundingSplitBps` percentage can be changed.

The issue is that the `_finalize` function can only succeed when the fees can be transferred to the recipient.

However the recipient contract may revert when it receives ETH. This causes all ETH in the `ETHCrowdfundBase` contract to be stuck.

Proof of Concept

When the crowdfund is won the `finalize` function needs to be called which calls `_finalize`:

Link

```
function _finalize(uint96 totalContributions_) internal {
    // Finalize the crowdfund.
    delete expiry;


    // Update the party's total voting power.
    uint96 newVotingPower = (totalContributions_ * exchangeRateI
    party.increaseTotalVotingPower(newVotingPower);
```

```
        // Transfer fee to recipient if applicable.
        address payable fundingSplitRecipient_ = fundingSplitRecipie
        uint16 fundingSplitBps_ = fundingSplitBps;
        if (fundingSplitRecipient_ != address(0) && fundingSplitBps_
            uint96 feeAmount = (totalContributions_ * fundingSplitBp
            totalContributions_ -= feeAmount;
            fundingSplitRecipient_.transferEth(feeAmount);
        }


        // Transfer ETH to the party.
        payable(address(party)).transferEth(totalContributions_);
    }
```

Here you can see that the `feeAmount` is transferred to the `fundingSplitRecipient`:

```
    fundingSplitRecipient_.transferEth(feeAmount);
```

If the recipient contract reverts, the ETH cannot be transferred and the crowdfund cannot be finalized.

But the users can also not get a refund because the crowdfund is in the `Won` state. So there is no way to get the funds out of the contract which means they are lost. Also the users don't get the voting power that they are supposed to get from the crowdfund.

This could be used in a griefing attack where the `fundingSplitRecipient` is set such that it can be made to revert.

Users that fall into this "trap" will lose all their funds. It can also just happen by mistake that a bad `fundingSplitRecipient` is set.

Tools Used

VSCode

Recommended Mitigation Steps

I recommend to pay the fees in a separate function such that it is separated from the `_finalize` function.

```
diff --git a/contracts/crowdfund/ETHCrowdfundBase.sol b/contract
index 4392655..5f68406 100644
--- a/contracts/crowdfund/ETHCrowdfundBase.sol
+++ b/contracts/crowdfund/ETHCrowdfundBase.sol
@@ -62,6 +62,8 @@ contract ETHCrowdfundBase is Implementation {
     error BelowMinimumContributionsError(uint96 contributions,
     error AboveMaximumContributionsError(uint96 contributions,
     error ContributingForExistingCardDisabledError();
+    error NotFinalizedError();
+    error FundingFeesAlreadyPaidError();

     event Contributed(
         address indexed sender,
@@ -109,6 +111,8 @@ contract ETHCrowdfundBase is Implementation
     /// @notice The address a contributor is delegating their v
     mapping(address => address) public delegationsByContributor

+    bool public fundingFeesPaid;
+
     // Initialize storage for proxy contracts, credit initial c
     // any), and setup gatekeeper.
     function _initialize(ETHCrowdfundOptions memory opts) inter
@@ -278,7 +282,20 @@ contract ETHCrowdfundBase is Implementatior
         uint96 newVotingPower = (totalContributions_ * exchange
         party.increaseTotalVotingPower(newVotingPower);

+        // Transfer ETH to the party.
+        payable(address(party)).transferEth(totalContributions_
+    }
+
+    function sendFundingFees() external {
+        CrowdfundLifecycle lc = getCrowdfundLifecycle();
+
+        if (lc != CrowdfundLifecycle.Finalized) revert NotFinal
+        if (fundingFeesPaid) revert FundingFeesAlreadyPaidError
+
+        fundingFeesPaid = true;
+
         // Transfer fee to recipient if applicable.
+        uint96 totalContributions_ = totalContributions;
```

```
            address payable fundingSplitRecipient_ = fundingSplitRe
            uint16 fundingSplitBps_ = fundingSplitBps;
            if (fundingSplitRecipient_ != address(0) && fundingSpli
@@ -286,8 +303,5 @@ contract ETHCrowdfundBase is Implementation
                totalContributions_ -= feeAmount;
                fundingSplitRecipient_.transferEth(feeAmount);
            }
-
-            // Transfer ETH to the party.
-            payable(address(party)).transferEth(totalContributions_
        }
    }
```

Alternatively it may also be an option to just send the fees to the party in case the transfer to the recipient fails.

0xble (Party) confirmed

[M-09] `InitialETHCrowdfund` + `ReraiseETHCrowdfund`: Gatekeeper checks wrong address

Submitted by **HollaDieWaldfee**

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/InitialETHCrowdfund.sol#L282-L293

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ReraiseETHCrowdfund.sol#L215-L226

This vulnerability exists in both the `InitialETHCrowdfund` and `ReraiseETHCrowdfund` contracts in exactly the same way.

I will continue this report by explaining the issue in only one contract. The mitigation section however contains the fix for both instances.

When making a contribution there is a check with the `gatekeeper` (if it is configured):

Link

```
    // Must not be blocked by gatekeeper.
    IGateKeeper _gateKeeper = gateKeeper;
    if (_gateKeeper != IGateKeeper(address(0))) {
        if (!_gateKeeper.isAllowed(contributor, gateKeeperId, gateDa
            revert NotAllowedByGateKeeperError(
                contributor,
                _gateKeeper,
                gateKeeperId,
                gateData
            );
        }
    }
```

The issue is that the first argument to the `isAllowed` function is wrong. It is `contributor` but it should be `msg.sender`.

The impact of this is that it will be possible for unauthorized users to make contributions.

Proof of Concept

Fortunately the new `InitialETHCrowdfund` and `ReraiseETHCrowdfund` contracts are very similar to the already audited other crowdfund contracts.

So we can look into the `Crowdfund.sol` code and see how the `gatekeeper.isAllowed` function should be called:

Link

```
        function _contribute(
            address contributor,
            address delegate,
            uint96 amount,
            uint96 previousTotalContributions,
```

```
        bytes memory gateData
    ) private {
        if (contributor == address(this)) revert InvalidContribu


        if (amount == 0) return;


        // Must not be blocked by gatekeeper.
        {
            IGateKeeper _gateKeeper = gateKeeper;
            if (_gateKeeper != IGateKeeper(address(0))) {
                if (!_gateKeeper.isAllowed(msg.sender, gateKeepe
                    revert NotAllowedByGateKeeperError(
                        msg.sender,
                        _gateKeeper,
                        gateKeeperId,
                        gateData
                    );
                }
            }
        }
```

We can see that the first argument to the `gatekeeper.isAllowed` function is `msg.sender`.

This means that when User A contributes for User B, the address that is checked is the address of User A and not the address of User B.

The new crowdfund contracts however check `contributor`:

Link

```
    // Must not be blocked by gatekeeper.
    IGateKeeper _gateKeeper = gateKeeper;
    if (_gateKeeper != IGateKeeper(address(0))) {
        if (!_gateKeeper.isAllowed(contributor, gateKeeperId, gateDa
            revert NotAllowedByGateKeeperError(
                contributor,
                _gateKeeper,
                gateKeeperId,
                gateData
```

```
            );
        }
    }
```

This means that when User A contributes for User B, the address of User B is checked. However it should be the address of User A (as seen above).

Imagine a situation where three addresses are whitelisted by the gatekeeper:

```
Alice
Bob
Chris
```

What should be checked by the gatekeeper is that only Alice, Bob and Chris can make contributions but they should be able to make contributions for everyone (check `msg.sender` instead of contributor).

What is actually checked is that any user can make a contribution but they can only contribute to Alice, Bob and Chris (check contributor instead of `msg.sender`).

Tools Used

VSCode

Recommended Mitigation Steps

In both contracts the `msg.sender` needs to be checked instead of `contributor`.

```
diff --git a/contracts/crowdfund/InitialETHCrowdfund.sol b/contr
index 8ab3b5c..fa8ec5d 100644
--- a/contracts/crowdfund/InitialETHCrowdfund.sol
+++ b/contracts/crowdfund/InitialETHCrowdfund.sol
@@ -282,9 +282,9 @@ contract InitialETHCrowdfund is ETHCrowdfunc
         // Must not be blocked by gatekeeper.
         IGateKeeper _gateKeeper = gateKeeper;
         if (_gateKeeper != IGateKeeper(address(0))) {
-            if (!_gateKeeper.isAllowed(contributor, gateKeeperI
+            if (!_gateKeeper.isAllowed(msg.sender, gateKeeperId
                 revert NotAllowedByGateKeeperError(
```

```
-                   contributor,
+                   msg.sender,
                    _gateKeeper,
                    gateKeeperId,
                    gateData



diff --git a/contracts/crowdfund/ReraiseETHCrowdfund.sol b/contr
index 580623d..72f3a20 100644
--- a/contracts/crowdfund/ReraiseETHCrowdfund.sol
+++ b/contracts/crowdfund/ReraiseETHCrowdfund.sol
@@ -215,9 +215,9 @@ contract ReraiseETHCrowdfund is ETHCrowdfunc
         // Must not be blocked by gatekeeper.
         IGateKeeper _gateKeeper = gateKeeper;
         if (_gateKeeper != IGateKeeper(address(0))) {
-            if (!_gateKeeper.isAllowed(contributor, gateKeeperI
+            if (!_gateKeeper.isAllowed(msg.sender, gateKeeperId
                 revert NotAllowedByGateKeeperError(
-                    contributor,
+                    msg.sender,
                     _gateKeeper,
                     gateKeeperId,
                     gateData
```

On this note it is important to mention that there is also an issue in `Crowdfund.sol` which is out of scope but the issue is of importance here:

The issue is in the `Crowdfund.batchContributeFor` function.

The function calls `this.contributeFor` **Link**.

So when the call is made, `msg.sender` is the address of the crowdfund and not the address of the user.

Therefore the gatekeeper check is wrong **Link**.

This is clearly not how the gatekeeper should be used. The gatekeeper should check the address of the user.

If you implement in the `ReraiseETHCrowdfund` and `InitialETHCrowdfund` contracts the changes I suggested, the same issue will be introduced there.

The solution is to call `_contributeFor` directly and to remove the `revertOnFailure` option. Or do a more involved change with supplying the correct `msg.sender`.

Oxean (judge) commented:

> Will leave open for sponsor review.

> I think the crux of it comes down to this assumption by the warden:

> "This is clearly not how the gatekeeper should be used. The gatekeeper should check the address of the user."

> I believe the design is meant to check instead for the address of the `contributor` but cannot find that explicitly documented.

0xble (Party) confirmed and commented:

> I think it makes the most sense to be consistent with the behavior of prior crowdfunds here. Will implement the recommended mitigation.

> The note about `Crowdfund.batchContributeFor` is also interesting and something we'll look into. Very glad it was brought up even though it was out of scope.

[M-10] `OperatorProposal.sol` : Leftover ETH is not refunded to the `msg.sender`

Submitted by **HollaDieWaldfee**, *also found by* **hansfriese**

The `OperatorProposal` contract is a type of proposal that allows to execute operations on contracts that implement the `IOperator` interface.

Upon execution of the proposal it might be necessary that the `executor` provides ETH.

This is true especially when `allowOperatorsToSpendPartyEth=false`, i.e. when ETH cannot be spent from the Party's balance. So it must be provided by the `executor`.

The amount of ETH that is needed to execute the operation is sent to the operator contract:

Link

```
data.operator.execute{ value: data.operatorValue }(data.operator
```

The operator contract then spends whatever amount of ETH is actually necessary and returns the remaining ETH.

For example the `CollectionBatchBuyOperator` contract may not spend all of the ETH because the actual purchases that are made are not necessarily known at the time the proposal is created. Also not all purchases may succeed.

So it is clear that some of the ETH may be returned from the `operator` to the `OperatorProposal` contract.

The issue is that the remaining ETH is not refunded to the `executor` and therefore this results in a direct loss of funds for the `executor`.

I discussed this issue with the sponsor and it is clear that the remaining ETH needs to be refunded when `allowOperatorsToSpendPartyEth=false`.

However it is not clear what to do when `allowOperatorsToSpendPartyEth=true`. In this case ETH can be spent from the party's balance. So there should be limited use cases for the `executor` providing additional ETH.

But if the `executor` provides additional ETH what should happen?

Should the ETH be taken from the `executor` first? Or should it be taken from the Party balance first?

The sponsor mentioned that since there are limited use cases for the `executor` providing additional ETH it may be ok to not refund ETH at all.

I disagree with this. Even when `allowOperatorsToSpendPartyEth=true` there should be a policy for refunds. I.e. the necessary ETH should either be taken from the Party's balance or from the `executor` first and any remaining funds from the `executor` should be returned.

However since it is not clear how to proceed in this case and since it is less important compared to the case where `allowOperatorsToSpendPartyEth=false` I will only make a suggestion for the case where `allowOperatorsToSpendPartyEth=false`.

The sponsor should decide what to do in the other case and make the appropriate changes.

Proof of Concept

When the `executor` executes an `OperatorProposal`, `operatorValue` amount of ETH is sent to the `operator` contract (when `allowOperatorsToSpendPartyEth=false` all of these funds must come from the `msg.value`):

Link

```
    if (!allowOperatorsToSpendPartyEth && data.operatorValue > msg.v
        revert NotEnoughEthError(data.operatorValue, msg.value);
    }


    // Execute the operation.
    data.operator.execute{ value: data.operatorValue }(data.operator
```

Currently the only `operator` contract that is implemented is the `CollectionBatchBuyOperator` and as explained above not all of the funds may be used so the funds are sent back to the Party:

Link

```
    uint256 unusedEth = msg.value - totalEthUsed;
    if (unusedEth > 0) payable(msg.sender).transferEth(unusedEth);
```

However after calling the `operator` contract, the `OperatorProposal` contract just returns without sending back the unused funds to the `executor` (`msg.sender`).

Link

```
    // Nothing left to do.
    return "";
```

So there is a loss of funds for the `executor`. The leftover funds are effectively transferred to the Party.

Tools Used

VSCode

Recommended Mitigation Steps

As mentioned before, this is only a fix for the case when `allowOperatorsToSpendPartyEth=false`.

Fix:

```
    diff --git a/contracts/proposals/OperatorProposal.sol b/contract
    index 23e2897..507e0d5 100644
    --- a/contracts/proposals/OperatorProposal.sol
    +++ b/contracts/proposals/OperatorProposal.sol
    @@ -4,7 +4,11 @@ pragma solidity 0.8.17;
     import "./IProposalExecutionEngine.sol";
```

```
 import "../operators/IOperator.sol";

+import "../utils/LibAddress.sol";
+
 contract OperatorProposal {
+    using LibAddress for address payable;
+
     struct OperatorProposalData {
         // Addresses that are allowed to execute the proposal a
         // calldata used by the operator proposal at the time c
@@ -41,9 +45,17 @@ contract OperatorProposal {
             revert NotEnoughEthError(data.operatorValue, msg.va
         }

+        uint256 partyBalanceBefore = address(this).balance - ms
+
         // Execute the operation.
         data.operator.execute{ value: data.operatorValue }(data

+        if (!allowOperatorsToSpendPartyEth) {
+            if (address(this).balance - partyBalanceBefore > 0)
+                payable(msg.sender).transferEth(address(this).b
+            }
+        }
+
         // Nothing left to do.
         return "";
     }
```

[Oxean (judge) commented]():

> Looking forward to sponsor comment on this. As described in my comment on #30 , I am not entirely sure that this qualifies as Medium based on the fact that the caller presumably is able to call with the correct value. QA may be more appropriate.

[HollaDieWaldfee (warden) commented]():

> @Oxean - It is not true that the caller can just use the function with the correct value:

> “For example the CollectionBatchBuyOperator contract may not spend all of the ETH because the actual purchases that are made are not necessarily known at the time the proposal is created. Also not all purchases may succeed.”

0xble (Party) confirmed and commented:

> @HollaDieWaldfee is right, there is at least one case with the `CollectionBatchBuyOperator` where it may not use all the ETH and the user may expect to be refunded. I think marking as Medium is valid.

[M-11] `CollectionBatchBuyOperator.sol : tokenIds` array is not shortened properly which makes execute function revert when not all NFTs are purchased successfully

Submitted by **HollaDieWaldfee**, *also found by* **hansfriese**

The `CollectionBatchBuyOperator` contract allows parties to buy NFTs through proposals.

The proposal specifies an `nftContract` and token IDs (via the `nftTokenIdsMerkleRoot` parameter) that can be bought.

Allowed `executors` can then execute the actual purchase by executing the proposal and providing execution data.

The execution data specifies which token IDs to buy, where to buy them from and the price to buy the tokens for.

The `CollectionBatchBuyOperator.execute` function is supposed to succeed even when not all purchases are successful.

This is achieved by skipping over failed purchases:

Link

```
    {
        // Execute the call to buy the NFT.
```

```
        (bool success, ) = _buy(call.target, callValue, call.data);


        if (!success) continue;
    }
```

Later in the function the NFTs that have been bought are transferred to the party:

Link

```
    for (uint256 i; i < tokenIds.length; ++i) {
        op.nftContract.safeTransferFrom(address(this), msg.sender, t
    }
```

If at least one NFT purchase has failed, the `tokenIds` array is bigger than the amount of NFTs that has actually been purchased. In other words there are empty spots at the end of the `tokenIds` array, i.e. the value that is stored there is zero.

Therefore, before transferring the NFTs, the `tokenIds` array needs to be shortened such that it is not attempted to transfer `tokenId=0`.

The contract uses the following code to achieve this:

Link

```
    assembly {
        // Update length of `tokenIds`
        mstore(mload(ex), tokensBought)
    }
```

This code is wrong as I will explain later.

The impact of this is that when not all purchases are successful the function reverts because it attempts to transfer the `tokenId=0` (since there are empty spots in the `tokenIds` array and the array is not shortened).

So the execution of the proposal will fail when it should actually succeed.

Proof of Concept

Let's have a look again at the code to shorten the `tokenIds` array:

[Link]

```
    assembly {
        // Update length of `tokenIds`
        mstore(mload(ex), tokensBought)
    }
```

It loads the first 32 bytes of `ex` from memory (`ex` is a `CollectionBatchBuyExecutionData` struct) and stores `tokensBought` in the memory location where the 32 bytes point to.

This has nothing to do with shortening the `tokenIds` array.

The correct code would be:

```
    assembly {
        // Update length of `tokenIds`
        mstore(tokenIds, tokensBought)
    }
```

This writes `tokensBought` to the first 32 bytes slot of the `tokenIds` array which is where the size of the array is stored.

There exists a test case for this scenario in the `CollectionBatchBuyOperator.t.sol` test file. However the test contains an error which makes the test pass even though the `tokenIds` array is not shortened.

Apply these changes to the test file:

```
diff --git a/sol-tests/operators/CollectionBatchBuyOperator.t.sc
```

```
index 3956e84..a944c74 100644
--- a/sol-tests/operators/CollectionBatchBuyOperator.t.sol
+++ b/sol-tests/operators/CollectionBatchBuyOperator.t.sol
@@ -165,7 +165,7 @@ contract CollectionBatchBuyOperatorTest is T
         bytes memory executionData = abi.encode(
             CollectionBatchBuyOperator.CollectionBatchBuyExecut
                 calls: calls,
-                numOfTokens: 2
+                numOfTokens: 3
             })
         );
```

Notice that when running the test (with the changes to the test file applied) it fails since the `tokenIds` array is not shortened properly.

Then also apply the changes to the source file (shortening the array properly) and see that the test passes.

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

As explained above, this is how to properly shorten the `tokenIds` array:

```
diff --git a/contracts/operators/CollectionBatchBuyOperator.sol
index 4b1dcc9..fffa5e9 100644
--- a/contracts/operators/CollectionBatchBuyOperator.sol
+++ b/contracts/operators/CollectionBatchBuyOperator.sol
@@ -179,7 +179,7 @@ contract CollectionBatchBuyOperator is IOper

         assembly {
             // Update length of `tokenIds`
-            mstore(mload(ex), tokensBought)
+            mstore(tokenIds, tokensBought)
         }
```

Oxble (Party) confirmed

[M-12] `VetoProposal` : proposals cannot be vetoed in all states in which it should be possible to veto proposals

Submitted by HollaDieWaldfee*, also found by* hansfriese

The `VetoProposal` contract allows to veto proposals with the `voteToVeto` function.

The proposal can only be vetoed when it is in the `Voting` state, otherwise the `voteToVeto` function reverts.

The issue is that the `Voting` state is not the only state in which it should be possible to veto the proposal. It should also be possible to veto the proposal in the `Passed` and `Ready` states.

(We can see this by looking at the downstream `PartyGovernance.veto` function)

It has been confirmed to me by the sponsor that the `voteToVeto` function should not restrict the situations in which vetos can occur.

The impact of this issue is that the situations in which vetos can occur is more limited than it should be. Users should have the ability to veto proposals even in the `Passed` and `Ready` states but they don't.

Proof of Concept

By looking at the `VetoProposal.voteToVeto` function we see that it's only possible to call the function when the proposal is in the `Voting` state. Otherwise the function reverts:

Link

```
    // Check that proposal is active
    (
        PartyGovernance.ProposalStatus proposalStatus,
        PartyGovernance.ProposalStateValues memory proposalValues
    ) = party.getProposalStateInfo(proposalId);
    if (proposalStatus != PartyGovernance.ProposalStatus.Voting)
```

```
        revert ProposalNotActiveError(proposalId);
```

But when we look at the `PartyGovernance.veto` function which is called downstream and which implements the actual veto functionality (the `VetoProposal.voteToVeto` function is only a wrapper) we can see that it allows vetoing in the `Voting`, `Passed` and `Ready` states:

Link

```
    function veto(uint256 proposalId) external onlyHost onlyDelegate
        // Setting `votes` to -1 indicates a veto.
        ProposalState storage info = _proposalStateByProposalId[prop
        ProposalStateValues memory values = info.values;


        {
            ProposalStatus status = _getProposalStatus(values);
            // Proposal must be in one of the following states.
            if (
                status != ProposalStatus.Voting &&
                status != ProposalStatus.Passed &&
                status != ProposalStatus.Ready
            ) {
                revert BadProposalStatusError(status);
            }
        }


        // -1 indicates veto.
        info.values.votes = VETO_VALUE;
        emit ProposalVetoed(proposalId, msg.sender);
    }
```

Therefore we can see that the `VetoProposal.voteToVeto` function restricts the vetoing functionality too much.

Users are not able to veto in the `Passed` and `Ready` states even though it should be possible.

Tools Used

VSCode

Recommended Mitigation Steps

The issue can be fixed by allowing the `VetoProposal.voteToVeto` function to be called in the `Passed` and `Ready` states as well.

Fix:

```
diff --git a/contracts/proposals/VetoProposal.sol b/contracts/pr
index 780826f..38410f6 100644
--- a/contracts/proposals/VetoProposal.sol
+++ b/contracts/proposals/VetoProposal.sol
@@ -30,7 +30,11 @@ contract VetoProposal {
             PartyGovernance.ProposalStatus proposalStatus,
             PartyGovernance.ProposalStateValues memory proposal
         ) = party.getProposalStateInfo(proposalId);
-        if (proposalStatus != PartyGovernance.ProposalStatus.Vo
+        if (
+            proposalStatus != PartyGovernance.ProposalStatus.Vo
+            && proposalStatus != PartyGovernance.ProposalStatus
+            && proposalStatus != PartyGovernance.ProposalStatus
+          )
             revert ProposalNotActiveError(proposalId);
```

0xble (Party) confirmed and commented via duplicate issue `#20`:

> This is valid, proposals should be allowed to be vetoed even after they've passed but have not yet been executed.

Low Risk and Non-Critical Issues

For this audit, 3 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **HollaDieWaldfee** received the top score from the judge.

The following wardens also submitted reports: **evan** *and* **hansfriese**.

Summary

Risk	Title	File	Instances
L-01	ETH is not refunded when `allowArbCallsToSpendPatyETH=true`	ArbitraryCallsProposal.sol	1
L-02	Comments state that pre-existing ETH can be used but it can't	-	2
L-03	Issue due to rounding from previous C4 audit is still present in new crowdfund contracts	-	2
L-04	Use `delegationsByContributor[contributor]` instead of `delegate` when minting party card	InitialETHCrowdfund.sol	1
L-05	Attacker can decide how voting power is distributed across party cards (griefing attack)	ReraiseETHCrowdfund.sol	1
L-06	Use `uint256` for computations such that voting power can be all values in `uint96` range	PartyGovernance.sol	1
L-07	Allow specifying `maximumPrice` for individual NFTs	CollectionBatchBuyOperator.sol	1
N-01	Introduce separate `vetoThresholdBps` for vetoing a proposal	VetoProposal.sol	1
N-02	`OperationExecuted` event is defined but never emitted	OperatorProposal.sol	1
N-03	Use `transferEth` instead of `transfer` for transferring ETH	-	4
N-04	Check that none of the `authorities` is zero address	PartyFactory.sol	1

[L-01] ETH is not refunded when `allowArbCallsToSpendPatyETH=true`

The `ArbitraryCallsProposal` contract does not refund `ETH` to the `msg.sender` if `allowArbCallsToSpendPartyEth=true`.

It only refunds when `allowArbCallsToSpendPartyEth=false`:

Link

```
    if (!allowArbCallsToSpendPartyEth && ethAvailable > 0) {
        payable(msg.sender).transferEth(ethAvailable);
    }
```

The reason this is the case is that it is not expected that the `msg.sender` will provide `ETH` if it is allowed to spend `ETH` from the Party's balance.

I don't think this is a good assumption. There should be a refund mechanism.

Please also refer to my report `#5` which discusses a more severe similar issue. Some of the reasoning there also applies here. Specifically that there are two broad options to implement refunds when `allowArbCallsToSpendPartyEth=true`.

1. Use `msg.value` first
2. Use Party balance first

The sponsor needs to decide which policy (if any) to use.

[L-02] Comments state that pre-existing ETH can be used but it can't

It is possible to provide an initial contribution to the `ReraiseETHCrowdfund` and `InitialETHCrowdfund` contracts.

The initial contribution is processed when the `initialize` function is called.

In both contracts it is stated that pre-existing ETH is used for the initial contribution (i.e. ETH that is owned by the contract but not sent along with the call to the `initialize` function):

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/InitialETHCrowdfund.sol#L118-L131

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ReraiseETHCrowdfund.sol#L80-L87

However the initial contribution is only measured by looking at `msg.value`:

Link

```
uint96 initialContribution = msg.value.safeCastUint256ToUint96()
```

This means that pre-existing ETH is not actually processed and just sits in the contract without being used for anything. It can't even be rescued.

It was assessed with the sponsor that they don't actually want to allow pre-existing ETH to be used for the initial contribution.

Therefore the comments should be removed so users don't make a mistake and lose their ETH.

Fix:

```
diff --git a/contracts/crowdfund/InitialETHCrowdfund.sol b/contr
index 8ab3b5c..bcc65e2 100644
--- a/contracts/crowdfund/InitialETHCrowdfund.sol
+++ b/contracts/crowdfund/InitialETHCrowdfund.sol
@@ -115,12 +115,9 @@ contract InitialETHCrowdfund is ETHCrowdfur
             })
         );

-        // If the deployer passed in some ETH during deployment
-        // for the initial contribution.
         uint96 initialContribution = msg.value.safeCastUint256T
         if (initialContribution > 0) {
-            // If this contract has ETH, either passed in durir
-            // pre-existing, credit it to the `initialContribut
+            // credit msg.value to the `initialContributor`.
             _contribute(
                 crowdfundOpts.initialContributor,
                 crowdfundOpts.initialDelegate,
```

```
diff --git a/contracts/crowdfund/ReraiseETHCrowdfund.sol b/contr
index 580623d..6ad81fc 100644
--- a/contracts/crowdfund/ReraiseETHCrowdfund.sol
```

```
+++ b/contracts/crowdfund/ReraiseETHCrowdfund.sol
@@ -77,12 +77,9 @@ contract ReraiseETHCrowdfund is ETHCrowdfundB
             0 // Ignored. Will use customization preset from pa
         );

-        // If the deployer passed in some ETH during deployment
-        // for the initial contribution.
         uint96 initialContribution = msg.value.safeCastUint256T
         if (initialContribution > 0) {
-            // If this contract has ETH, either passed in durir
-            // pre-existing, credit it to the `initialContribut
+            // credit msg.value to the `initialContributor`.
             _contribute(opts.initialContributor, opts.initialDe
         }
```

[L-03] Issue due to rounding from previous C4 audit is still present in new crowdfund contracts

In the previous C4 audit an issue has been found that due to rounding it might not be possible to achieve an unanimous vote:

https://code4rena.com/reports/2022-09-party/#m-10-possible-that-unanimous-votes-is-unachievable

The issue also exists in the new crowdfund contracts (`InitialETHCrowdfund` , `ReraiseETHCrowdfund`).

I agree that it would be best to introduce the fix suggested by the warden in the old report.

However the sponsor told me that the `minContribution` amount will be set large enough such that the rounding issue cannot occur.

Therefore I don't think this issue is worth reporting as Medium in this audit again. But I'd like to bring it up because it's still present in the new contracts.

[L-04] Use `delegationsByContributor[contributor]` instead of `delegate` when minting party card

The `ÌnitialETHCrowdfund._contribute` function currently uses `delegate` as the delegate when minting a party card Link.

This is wrong. Instead `delegationsByContributor[contributor]` should be used. This is because if User A contributes for another User B and User B has already set a delegate, i.e. `delegationsByContributor[contributor]!=address(0)` then this delegate should be used and not the `delegate` parameter supplied by User A.

However I don't see any impact in this behavior because if `delegationsByContributor[contributor]!=address(0)` then the party also has a delegate set for the `contributor`. And so this delegate is used above the delegate from the crowdfund anyway. So this finding is Low severity at most.

Fix:

```
diff --git a/contracts/crowdfund/InitialETHCrowdfund.sol b/contr
index 8ab3b5c..6c76e88 100644
--- a/contracts/crowdfund/InitialETHCrowdfund.sol
+++ b/contracts/crowdfund/InitialETHCrowdfund.sol
@@ -299,7 +299,7 @@ contract InitialETHCrowdfund is ETHCrowdfunc

         if (tokenId == 0) {
             // Mint contributor a new party card.
-            party.mint(contributor, votingPower, delegate);
+            party.mint(contributor, votingPower, delegationsByC
         } else if (disableContributingForExistingCard) {
             revert ContributingForExistingCardDisabledError();
         } else if (party.ownerOf(tokenId) == contributor) {
```

[L-05] Attacker can decide how voting power is distributed across party cards (griefing attack)

In the `ReraiseETHCrowdfund` contract the party cards are not minted immediately. The user first gets a crowdfund NFT and later when the crowdfund is won the voting power can be "claimed" which means the actual party cards are minted.

Voting power is claimed with the `ReraiseETHCrowdfund.claim` or `ReraiseETHCrowdfund.claimMultiple` function.

The issue arises from the fact that any User A can claim the party cards of any other User B.
And User A can also decide how the voting power is distributed across party cards (within the limits set in the contract).

Also party cards cannot be reorganized. When a party card has voting power 10 it has voting power 10 forever. It is not possible to divide this party card into two party cards with voting power 5 each.
Similarly two party cards cannot be merged into one.

From the above observations we can understand how this leads to a problem: A User A may want to have 10 party cards with 1 voting power each so he can transfer them individually if needed. The attacker can do a griefing attack and claim 1 party card with voting power 10 so the party card cannot be used as intended.

The sponsor explained that it is important that anyone can claim party cards so this is not something we can restrict.

Also I have been told that they have considered for while to allow reorganizing party cards.
Therefore I encourage them to actually implement reorganizing party cards.

Thereby it is ensured that if this griefing attack occurs a user can reorganize his party cards and use them as intended.

[L-06] Use `uint256` for computations such that voting power can be all values in `uint96` range

The voting power in a party is managed in `uint96` variables.

When we look at the `PartyGovernance._areVotesPassing` function we can see that for the computation the `uint96` variables are cast to `uint256`:

Link

```
    function _areVotesPassing(
        uint96 voteCount,
        uint96 totalVotingPower,
```

```
        uint16 passThresholdBps
    ) private pure returns (bool) {
        return (uint256(voteCount) * 1e4) / uint256(totalVotingPower
    }
```

This is done such that there is no intermediate overflow. If there was no cast, the multiplication `voteCount * 1e4` could overflow and cause a DOS to the Party when `voteCount` is close to `type(uint96).max = ~ 7.9e28`.

The issue is in the `PartyGovernance._isUnanimousVotes` function which does not convert the voting power to `uint256` and is therefore prone to overflow:

Link

```
    function _isUnanimousVotes(
        uint96 totalVotes,
        uint96 totalVotingPower
    ) private pure returns (bool) {
        uint256 acceptanceRatio = (totalVotes * 1e4) / totalVotingPo
        // If >= 99.99% acceptance, consider it unanimous.
        // The minting formula for voting power is a bit lossy, so w
        // for slightly less than 100%.
        return acceptanceRatio >= 0.9999e4;
    }
```

As long as the mint authorities ensure that the number of votes stays within the safe range (`7.9e28 / 1e4 = ~7.9e24`) this is not a poblem. However the way the `_areVotesPassing` function works shows that the whole `uint96` range should be safe.

Therefore I propose the following change to the `_isUnanimousVotes` function:

```
diff --git a/contracts/party/PartyGovernance.sol b/contracts/par
index e251646..7571fa8 100644
--- a/contracts/party/PartyGovernance.sol
+++ b/contracts/party/PartyGovernance.sol
@@ -1038,7 +1038,7 @@ abstract contract PartyGovernance is
         uint96 totalVotes,
```

```
            uint96 totalVotingPower
        ) private pure returns (bool) {
-           uint256 acceptanceRatio = (totalVotes * 1e4) / totalVot
+           uint256 acceptanceRatio = (uint256(totalVotes) * 1e4) /
            // If >= 99.99% acceptance, consider it unanimous.
            // The minting formula for voting power is a bit lossy,
            // for slightly less than 100%.
```

[L-07] Allow specifying `maximumPrice` for individual NFTs

Currently it is only possible to specify a single `maximumPrice` for all NFTs in the `CollectionBatchBuyOperator` contract (Link).

I recommend that it should be possible to specify a `maximumPrice` for each NFT individually. Thereby it's possible for the party to enforce tighter limits to the amount of ETH that the executor can spend.

For example the NFT with `id=1` might require a maximum price of `1 ETH` whereas for another NFT with `id=2` a maximum price of `0.1 ETH` can be sufficient.

Currently both NFTs can only have the same maximum price and the executor is able to spend `2 ETH` at a maximum. This could be further restricted to minimize the trust that needs to be put into the executor.

This can be implemented by introducing a new `mapping(uint256 => uint256) maximumPrices` mapping. If the entry for an NFT is `!= 0` it should be used as the maximum price. Otherwise `maximumPrice` can be used as a fallback.

[N-01] Introduce separate `vetoThresholdBps` for vetoing a proposal

Currently the same `passThresholdBps` variable is used for accepting proposals as well as vetoing proposals.

`passTresholdBps` is a percentage of the `totalVotingPower` that is required.

I recommend to introduce a separate `vetoThresholdBps` governance parameter that is used to determine the percentage of votes necessary to veto a proposal.

Using separate thresholds allows for greater flexibility.

E.g. a Party might want a high consensus of 60% to accept a proposal but might want to require only 10% of votes to veto a proposal. Such a setup is not possible currently which unnecessarily restricts the flexibility of the protocol.

[N-02] `OperationExecuted` event is defined but never emitted

In the `OperatorProposal` contract the `OperationExecuted` event is defined but it is never emitted.

Therefore I recommend to emit this event when the operation is executed successfully.

Fix:

```
diff --git a/contracts/proposals/OperatorProposal.sol b/contract
index 23e2897..5899bec 100644
--- a/contracts/proposals/OperatorProposal.sol
+++ b/contracts/proposals/OperatorProposal.sol
@@ -44,6 +44,8 @@ contract OperatorProposal {
         // Execute the operation.
         data.operator.execute{ value: data.operatorValue }(data

+        emit OperationExecuted(msg.sender);
+
         // Nothing left to do.
         return "";
     }
```

[N-03] Use `transferEth` instead of `transfer` for transferring ETH

The automated findings have flagged the below instances as unsafe ERC20 operations.

This is wrong. They are not ERC20 operations. Instead they are just the `transfer` function that is a built-in Solidity function for sending ETH.

However the usage of this function is still **discouraged** because it limits the Gas that the callee can consume to `2300`.

Instead use the `transferEth` function that is used elsewhere in the codebase to transfer ETH.

There are 4 instances:
https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/InitialETHCrowdfund.sol#L204

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/InitialETHCrowdfund.sol#L267

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ReraiseETHCrowdfund.sol#L152

https://github.com/code-423n4/2023-04-party/blob/440aafacb0f15d037594cebc85fd471729bcb6d9/contracts/crowdfund/ReraiseETHCrowdfund.sol#L201

[N-04] Check that none of the `authorities` is zero address

The `PartyFactory.createParty` function should check that the `authorities` array contains addresses that are not the zero address.

I mention this because in the previous version of the `PartyFactory` contract, there was only one `authority` and it was checked to not be the zero address such as to ensure that governance NFTs can be minted:

Link

```
    if (authority == address(0)) {
        revert InvalidAuthorityError(authority);
    }
```

However now it is only checked that the `authorities` array is not empty:

Link

```
    if (authorities.length == 0) {
        revert NoAuthorityError();
    }
```

I think you should check that there is no zero address in this array. The current check is not sufficient when compared to what has been checked previously. The current check is weaker.

0xble (Party) confirmed

Gas Optimizations

For this audit, 2 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **HollaDieWaldfee** received the top score from the judge.

The following warden also submitted a report: hansfriese*.*

[G-01] Use `maxTotalContributations_` memory variable instead of `maxTotalContributions`

`maxTotalContributions` has already been saved to the `maxTotalContributions_` memory variable.
It's cheaper to read from memory than from storage.

Gas saved: 124 Gas per tx

```
diff --git a/contracts/crowdfund/ETHCrowdfundBase.sol b/contract
index 4392655..886c09c 100644
--- a/contracts/crowdfund/ETHCrowdfundBase.sol
+++ b/contracts/crowdfund/ETHCrowdfundBase.sol
@@ -212,7 +212,7 @@ contract ETHCrowdfundBase is Implementation
             _finalize(maxTotalContributions_);

             // Refund excess contribution.
-            uint96 refundAmount = newTotalContributions - maxTc
+            uint96 refundAmount = newTotalContributions - maxTc
             if (refundAmount > 0) {
                 amount -= refundAmount;
                 payable(msg.sender).transferEth(refundAmount);
```

[G-02] No need to cache `fundingSplitRecipient` in memory variable

`fundingSplitRecipient` is only used once. So it is not beneficial to cache it in a memory variable.

Gas saved: Negligible but code is simplified

```
diff --git a/contracts/crowdfund/ETHCrowdfundBase.sol b/contract
index 4392655..17af949 100644
--- a/contracts/crowdfund/ETHCrowdfundBase.sol
+++ b/contracts/crowdfund/ETHCrowdfundBase.sol
@@ -222,9 +222,8 @@ contract ETHCrowdfundBase is Implementation
         }

         // Subtract fee from contribution amount if applicable.
-        address payable fundingSplitRecipient_ = fundingSplitRe
         uint16 fundingSplitBps_ = fundingSplitBps;
-        if (fundingSplitRecipient_ != address(0) && fundingSpli
+        if (fundingSplitRecipient != address(0) && fundingSplit
             uint96 feeAmount = (amount * fundingSplitBps_) / 1e
             amount -= feeAmount;
         }
@@ -237,9 +236,8 @@ contract ETHCrowdfundBase is Implementation
         amount = (votingPower * 1e4) / exchangeRateBps;

         // Add back fee to contribution amount if applicable.
-        address payable fundingSplitRecipient_ = fundingSplitRe
```

```
            uint16 fundingSplitBps_ = fundingSplitBps;
-           if (fundingSplitRecipient_ != address(0) && fundingSpli
+           if (fundingSplitRecipient != address(0) && fundingSplit
                amount = (amount * 1e4) / (1e4 - fundingSplitBps_);
            }
        }
```

[G-03] Cache `party` in memory variable

`party` is used multiple times so it can be cached in a memory variable.

Gas saved: 149 per tx

```
diff --git a/contracts/crowdfund/InitialETHCrowdfund.sol b/contr
index 8ab3b5c..159e30a 100644
--- a/contracts/crowdfund/InitialETHCrowdfund.sol
+++ b/contracts/crowdfund/InitialETHCrowdfund.sol
@@ -325,15 +325,16 @@ contract InitialETHCrowdfund is ETHCrowdfu
         }

         // Get amount to refund.
-        uint96 votingPower = party.votingPowerByTokenId(tokenIc
+        Party party_ = party;
+        uint96 votingPower = party_.votingPowerByTokenId(tokenI
         amount = _calculateRefundAmount(votingPower);

         if (amount > 0) {
             // Get contributor to refund.
-            address payable contributor = payable(party.ownerOf
+            address payable contributor = payable(party_.ownerO

             // Burn contributor's party card.
-            party.burn(tokenId);
+            party_.burn(tokenId);

             // Refund contributor.
             contributor.transferEth(amount);
```

[G-04] Save `votingPowerByCard[i]` in memory variable

Gas saved: 1365 per tx

```
diff --git a/contracts/crowdfund/ReraiseETHCrowdfund.sol b/contr
index 580623d..68ccf4e 100644
--- a/contracts/crowdfund/ReraiseETHCrowdfund.sol
+++ b/contracts/crowdfund/ReraiseETHCrowdfund.sol
@@ -352,14 +352,15 @@ contract ReraiseETHCrowdfund is ETHCrowdfu
         uint96 minContribution_ = minContribution;
         uint96 maxContribution_ = maxContribution;
         for (uint256 i; i < votingPowerByCard.length; ++i) {
-            if (votingPowerByCard[i] == 0) continue;
+            uint96 vp = votingPowerByCard[i];
+            if (vp == 0) continue;

             // Check that the contribution equivalent of voting
             // contribution range. This is done so parties may
             // and maximum contribution values to limit the vot
             // card (e.g. a party desiring a "1 card = 1 vote"-
             // system where each card has equal voting power).
-            uint96 contribution = (votingPowerByCard[i] * 1e4)
+            uint96 contribution = (vp * 1e4) / exchangeRateBps;
             if (contribution < minContribution_) {
                 revert BelowMinimumContributionsError(contribut
             }
@@ -371,9 +372,9 @@ contract ReraiseETHCrowdfund is ETHCrowdfunc
             votingPower -= votingPowerByCard[i];

             // Mint contributor a new party card.
-            uint256 tokenId = party.mint(contributor, votingPow
+            uint256 tokenId = party.mint(contributor, vp, delec

-            emit Claimed(contributor, tokenId, votingPowerByCar
+            emit Claimed(contributor, tokenId, vp);
         }

         // Requires that all voting power is claimed because th
```

[G-05] Use `mintedVotingPower_` instead of `mintedVotingPower`

Gas saved: 114 per tx per instance

```
diff --git a/contracts/party/PartyGovernanceNFT.sol b/contracts/
index 9ccfa1f..c6c3d14 100644
```

```
--- a/contracts/party/PartyGovernanceNFT.sol
+++ b/contracts/party/PartyGovernanceNFT.sol
@@ -149,7 +149,7 @@ contract PartyGovernanceNFT is PartyGovernar

         // Update state.
         tokenId = tokenCount = tokenCount_ + 1;
-        mintedVotingPower += votingPower_;
+        mintedVotingPower = mintedVotingPower_ + votingPower_;
         votingPowerByTokenId[tokenId] = votingPower_;

         // Use delegate from party over the one set during crov
@@ -181,7 +181,7 @@ contract PartyGovernanceNFT is PartyGovernar
         }

         // Update state.
-        mintedVotingPower += votingPower_;
+        mintedVotingPower = mintedVotingPower_ + votingPower_;
         votingPowerByTokenId[tokenId] += votingPower_;

         _adjustVotingPower(ownerOf(tokenId), votingPower_.safeC
```

[0xble (Party) confirmed]

Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top