macro

PartyDAO A-1

Security Audit

October 31, 2022
Version 1.0.0

Presented by 0xMacro

Table of Contents

Introduction

This document includes the results of the security audit for PartyDAO's smart contract code as found in the section titled ‘Source Code’. The security audit was performed by the Macro security team from September 26, 2022 to October 21, 2022. And, is the 3rd audit on these contracts.

The purpose of this audit is to review the source code of certain PartyDAO Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro’s review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
High	1	-	-	1
Medium	2	-	-	2
Low	1	-	-	1
Code Quality	9	4	-	5
Informational	1	-	-	1
Gas Optimization	2	2	-	-

PartyDAO was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Telegram with the PartyDAO team.
- Available documentation in the repository.

Source Code

The following source code was reviewed during the audit:

- **Repository:** party-protocol
- **Commit Hash:** `ecd62753bfb1f001a111ba392cfca57b02beac77`

Specifically, we audited the following contracts within this repository:

Contract	SHA256
contracts/crowdfund/AuctionCrowdfund.sol	`98156f9a82a21c110fd43ff5c1f341e509047a6cb3d3ca74bae82758a3ca73c4`
contracts/crowdfund/BuyCrowdfund.sol	`51863984d1275284d6f81b593398803c9de6c080152c7fdeec622204d90967e7`
contracts/crowdfund/BuyCrowdfundBase.sol	`2adbc2415af1374d8e458b064d52e9ade78409cc7b84bf742713edcdf1c2094c`
contracts/crowdfund/CollectionBuyCrowdfund.sol	`793e15d104d5fa63babf7f5194a0654b5d6ec61b3a929d1128d66a15f659869f`
contracts/crowdfund/Crowdfund.sol	`5f8dfe0392b6d032f92950e26a445826cf564d597c6a65302d92380d47f5461e`
contracts/crowdfund/CrowdfundFactory.sol	`734befe53fef65a2729b6edef3948f03c983b8e84154bddcc80c5a64f130113a`
contracts/crowdfund/CrowdfundNFT.sol	`2608e7ce40758c8acf95b1cbd75e1ab31b6dcb734d67c7a664c0d81b684db95e`
contracts/distribution/ITokenDistributor.sol	`5b009aae28baf4f5773f199599a264014505d0992a90992adf56db7329d94741`

contracts/distribution/ITokenDistributorParty.sol	`61bd2a86a6c45ecf080a1f4df0f3dd41e517a2f1fe930c452cbd73aac710af9d`
contracts/distribution/TokenDistributor.sol	`505dc19ff3ed45da6dd3779c15b0d26fc988bf3de6abac0e40dfe6f33ef92f3b`
contracts/gatekeepers/IGateKeeper.sol	`60f646ff29f4e1023169e44b9284f8ee9051a75aeb05f5aa63224bf272101fa3`
contracts/globals/Globals.sol	`53e785d45dc0b1f110c10516b307b85ea456980b08a830b6d9bc6b28812859da`
contracts/globals/IGlobals.sol	`0a1c3ab36f45cf9ac178889e27ad9988819f07c046ec22bd67b288d317004d46`
contracts/market-wrapper/IMarketWrapper.sol	`14be10b7d082c9aac7c7b67ae96119bd64201f8c41430dd729890e83ccf41060`
contracts/party/IPartyFactory.sol	`0fda8be894eae7913c7b8b0d66b5f0196f53f39326afdc09d53eb2a4607b468a`
contracts/party/Party.sol	`2182e94562a474c80798f1b792f0cab1944dc4041931712b2196962417eade7b`
contracts/party/PartyFactory.sol	`e862b7873ac378231399a3bb0452ac0e917121203706add7118df14fba7ffe01`
contracts/party/PartyGovernance.sol	`1d9b6cbf337debf5fac5c879bcfe2576bfbdfff8006bab8d000cc98a15340b94`
contracts/party/PartyGovernanceNFT.sol	`45bac18b0696376499845499800f4bef5721985ae8b9b8890700ad89c1e21ade`
contracts/proposals/ArbitraryCallsProposal.sol	`a6c27b997340ec34ddbdd433fac70db4159ff082c96dfd942a9c6867590080b9`
contracts/proposals/FractionalizeProposal.sol	`14b9d3a7484a3e1b607c6f97026157a992b55ac1261ab4cd4aaeb10a416f340b`

contracts/proposals/IProposalExecutionEngine.sol	`7cf06db01ef962e96d6af0f0e2ab5224284f7ea8b38b25ac5bff415c54fb3def`
contracts/proposals/LibProposal.sol	`253783cc7c355dab07573aeeeb35d1c4ea7fcf5b9d444516a1e89ecb4deacedb`
contracts/proposals/ListOnOpenseaProposal.sol	`818c4b7c61de8207356e8803edeb361c590c043230506340b58d5a72948b498b`
contracts/proposals/ListOnZoraProposal.sol	`3e047c4558d4881e58fca24e738c9b0fc17e72501e91dc09893dc7f5b1a1615d`
contracts/proposals/ProposalExecutionEngine.sol	`3fbe004b4450872c60182bc7cb38852521c9dd567d700766405ec05ae6da137a`
contracts/proposals/ProposalStorage.sol	`35fcca7a918df1b2c9e02fb9aee90f4471183cf5f582f7df040c10e4186137de`
contracts/proposals/vendor/FractionalV1.sol	`7e66db4f0ca7e314ba11be8f2e961f76ec813b2cc64cd4e711412d24387b99d7`
contracts/proposals/vendor/IOpenseaConduitController.sol	`826d674fa14f75641d936f0d115ed21260dec391142d2892168770c60e257e72`
contracts/proposals/vendor/IOpenseaExchange.sol	`4d668f88d81debc16f76831b26be0296020a859d1a05630ad0e14636ad0157f9`
contracts/tokens/ERC1155Receiver.sol	`cd1867764528d50e5313d85132e68b95b86725ff900d34914648d3239f4ade04`
contracts/tokens/ERC721Receiver.sol	`47514904906ca19cc19a5ea7b2d91fdc18f7b66a6603cb26bcb50a5c57a04751`
contracts/tokens/IERC1155.sol	`9ee44456b0d4d0df22b63ccadaef549a8085ba551c9364ff3176059e10d2b5c3`
contracts/tokens/IERC20.sol	`0d89143c9c1c76d743db54bf50e91eaf2bf730b00c1051ca861bf9454c1129f0`

contracts/tokens/IERC721.sol	`df79ff7ee3f415fba621bd47a8c4a4b2cb0870ea24a28dc47767ec29e9eb9b7c`
contracts/tokens/IERC721Receiver.sol	`e6631daa9cae57e264cfec711ef2cfe051caff440e72911551cd9cfbedf75a57`
contracts/utils/EIP165.sol	`1e6c5a02010ed572745c0ba9ca939d9a8434d1547260409e2ec23b23b7aa6eba`
contracts/utils/Implementation.sol	`6b8b5b83636946f5ae9940d9b23fe2cc2bd9a30f7477b8b4e5dbdcc73acdf683`
contracts/utils/LibAddress.sol	`351cc04ce4cb9e6437d5ad8d51682805a1685eeb0552b8a8602e3cb0aaf08afc`
contracts/utils/LibERC20Compat.sol	`db468ae14f88bd4e3ada46fafa9d495969e378430b2b4a2725e0d4f4af53d165`
contracts/utils/LibRawResult.sol	`79c3a62d365678015c31d8a73505bcc305c76173a7dd86a5e4dc85d1d064be1f`
contracts/utils/LibSafeCast.sol	`844464129a8fb01899b4acd3ffc8fc28a5b67122f66567b835521a60f4d2eb41`
contracts/utils/LibSafeERC721.sol	`934be5ac56025ecdf1d70fec328d269315c20ba66608b158622ff8c0f92e2b12`
contracts/utils/Proxy.sol	`c298ec79b3942e7a4ab50c4f45821a6c530a7c407a762987fca31634e524c94a`
contracts/utils/ReadOnlyDelegateCall.sol	`3dbfb0b33f72c1a864ea156460f8aca537db2d5f5f3cdfdb03570c6dec302392`
contracts/vendor/markets/IZoraAuctionHouse.sol	`619b797882228eea5e426226e2f0b5395bc05514fb8703cb8b802ec33b481f39`

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:
    - How bad things can get (for a vulnerability)
    - The significance of an improvement (for a code quality issue)
    - The amount of gas saved (for a gas optimization)
2. The high/medium/low **likelihood** of the issue:
    - How likely is the issue to occur (for a vulnerability)
3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant funds/assets WILL be lost.**
(H-x) High	We recommend the client **must** address the issue, no matter what, because not fixing would be very bad, *or* some funds/assets will be lost, *or* the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to **seriously consider** fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albiet not in an existential manner.
(L-x) Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

~~H-1~~ The majority can steal precious NFT by combining Zora and arbitrary calls proposal

TOPIC	STATUS	IMPACT	LIKELIHOOD
Assets theft	Fixed	High ✳	Medium ✳

Added an indexCall and countCall. Can only call cancelAuction on Zora if it's the last call in the arbitrary proposal calls count.

Party allows the ZORA auction as one proposal type in their governance contracts.

Since ZORA is custodial if you cancel the proposal while the proposal is in `inprogress`, that NFT would remain inside the ZORA auction house.

Now, the party has two options: either wait for an auction to conclude, or call `cancelAuction()` 🔗 before the first bid is placed.

Party would need to execute an arbitrary proposal for the latter.

At this point only, the majority holder can trick the party into transferring that retrieved NFT outside of the party.

The ideal attack would look like this:

1. **Propose a `ListOnZoraProposal` with a high reserve price.**

```
Let voting happen, considering the attacker holds the majority and
there is no host intervention, and the proposal would pass.
```

2. **Execute ZORA Proposal.**

Proposal Step 1 : `createAuction()`

3. **`cancelProposal()`**

   The proposal is canceled, the party is unblocked for executing future proposals, and the NFT remains in the ZORA auction house.

4. **Propose an `ArbitraryCallsProposal` with two arbitrary calls.**

   Call 1: Call `cancelAuction()` on the auction house contract.

   Call 2: Transfer the NFT to an external address.
   `Let voting happen, considering the attacker holds the majority and there is no host intervention, and the proposal would pass.`

5. **Execute Arbitrary Calls Proposal**

   Since the reserve price is higher compared to the market value, no one would bid.

   Hence, `cancelAuction()` would succeed and return the NFT to the party.

   All is going as specifications expects, until here.

   Now, the issue is in the second call: **`Transferring NFT to an external address.`**

   Ideally, `ArbiartyCallsProposals` should fail on this call, as it is transferring precious NFT outside of the party since a check of ownership is done before and after the arbitrary calls.

```
        bool[] memory hadPreciouses = new bool[](params.preciousTokenIds.l
        if (!isUnanimous) {
```

```
            for (uint256 i = 0; i < hadPreciouses.length; ++i) {
                hadPreciouses[i] = _getHasPrecious(
                    params.preciousTokens[i],
                    params.preciousTokenIds[i]
                );
            }
        }
        <

            _executeSingleArbitraryCall(
                i,
                calls[i],
                params.preciousTokens,
                params.preciousTokenIds,
                isUnanimous,
                ethAvailable
            );

        >

        if (!isUnanimous) {
            for (uint256 i = 0; i < hadPreciouses.length; ++i) {
                if (hadPreciouses[i]) {
                    if (!_getHasPrecious(params.preciousTokens[i], params.
                        revert PreciousLostError(
                            params.preciousTokens[i],
                            params.preciousTokenIds[i]
                        );
                    }
                }
            }
        }
```

However, what if the party received the NFT in one of these single arbitrary calls?

The same thing happens as in the case of `cancelAuction` via arbitrary proposal.

Since the NFT is received in the middle, no record would be present in `hadPreciouses`.

Hence, the attacker can move the NFT outside via a second arbitrary call.

One simple solution is to move the `_getHasPrecious` check from multiple to single

arbitrary calls, but that would increase gas costs for all arbitrary call proposals; another solution is to block the Zora interactions on arbitrary calls.

~~M-1~~ Crowdfund contracts can be griefed into locking the target NFT after being purchased

TOPIC	STATUS	IMPACT	LIKELIHOOD
Griefing	Fixed	Medium ✳	Medium ✳

Added more checks for lastBid on AuctionCrowdfund. Removed the balance check from BuyCrowdfund contracts and now use only the callValue to assure was not gifted. The zero maximumPrice logic was removed, this was used to define a crowdfund without any max price. Still, the maximumPrice can be set to 0 creating a crowdfund that will revert always in any buy() call. This will cause contributions to get stuck in the contract until expiry.

On line 136, `CrowdfundBase.sol` has the following code inside the *_buy()* function logic:

```
{
    // Execute the call to buy the NFT.
    (bool s, bytes memory r) = callTarget.call{ value: callValue }(callDat
    if (!s) {
                r.rawRevert();
    }
}

if (token.safeOwnerOf(tokenId) == address(this)) {
    if (address(this).balance >= totalContributions_) {
        // If the purchase was free or the NFT was "gifted" to us,
        // refund all contributors by declaring we lost.
        settledPrice = 0;
        expiry = 0;
        emit Lost();
```

```
    } else {
        settledPrice = callValue;
        emit Won(
            // Create a party around the newly bought NFT.
            party_ = _createParty(
                governanceOpts,
                isValidatedGovernanceOpts,
                token,
                tokenId
            ),
            token,
            tokenId,
            callValue
        );
    }
```

There are two possible outcomes from this logic, considering that the crowdfunding contract is the owner of the target NFT (`token.safeOwnerOf(tokenId) == address(this)`):

- The balance of the contract is equal to or higher than the total contributions, and because of this, the contract will assume the NFT was gifted and will declare the crowdfunding as lost.
- Otherwise, the contract will call the crowdfunding as won, deploy the party, and transfer the NFT to the party.

In the first scenario, the contributors will be able to burn their crowdfunding NFT to reclaim their contributions, and the gifted NFT would be locked in the contract. Given that the contract only compares the balance to the `totalContributions` after the external `.call()`, a malicious actor could force send ether into the crowdfunding contract, pushing the balance higher than the `totalContributions`, even after the NFT is successfully purchased — essentially making the crowdfunding failed and the NFT locked.

This issue is also present in the `AuctionCrowdfund.sol` contract, and, given that this contract has a `receive()` function, the ether can even be sent by a regular

transaction.

⚠️ In a whitelisted sale, with a crowdfunding party being exclusively allowed to buy a specific NFT or buy from a specific collection, the attack vector could be considered higher since the party will assume the purchase of the token cannot fail. And an external actor can manipulate the results.

Resolution

One consideration in mitigating this would be to check the balance before and after executing the external call and compare this difference to the total contributions:

```
// Verify the actual balance of the contract
uint256 balanceBefore = address(this).balance;

// Execute the call to buy the NFT.
(bool s, bytes memory r) = callTarget.call{ value: callValue }(callData);

// Make sure we acquired the NFT we want.
if (token.safeOwnerOf(tokenId) == address(this)) {
    // The contract has more or the same balance than before
    if (balanceBefore <= address(this).balance) {
                                ...
                emit Lost();
            } else { // The contract spent eth
                settledPrice = callValue;
                emit Won();
            }
```

By doing this, the contract would ignore any ether outside contributions and assure that at least some funds have been used for the purchase. Still, this is just mitigation, since the external call could be force-sending ether to the contract, but it narrows the possible attack surface

---

~~M-2~~ Someone can grief an active crowdfund by advancing it to a `lost` state at a negligible cost

TOPIC	STATUS	IMPACT	LIKELIHOOD
Griefing	Fixed	Medium ✳	Medium ✳

Preventing approve and setApprovalForAll calls through buy() function. Only to the token address.

Let's consider that there is a non-gated `BuyCrowdFund` intended to buy one NFT. Multiple contributors are contributing, and all is going well.

Then, someone wants this crowdfund to fail, for some reason.

One accepted way for them to do this is to buy the given NFT. But they don't want to hold on to that NFT until expiry and don't have the funds to do so.

Another accepted way is to gift NFT to the crowdfund so that it would be locked forever, and crowdfund could be advanced to a `lost` state.

But they don't want to lose their funds.

What they can do?

Well, there is one way they can directly fail this crowdfund forever: moving it to a `lost` state, without losing ETH and without locking/gifting the NFT in crowdfund. They can:

1. Temporarily buy the NFT, and transfer it to the crowdfund contract.
2. Call `buy()` with `(callTarget=NFTContract, callData=approve(their address), callValue="0")`

   This results in the crowdfund contract approving the attacker's address as a spender for the NFT.

Since crowdfund owns the NFT at a null cost, the contract considers that the NFT is gifted and moves to a `lost` state.

Here, the assumption that NFT would remain locked forever so there is little incentive for the griefer to gift NFT won't hold true anymore

Since the NFT is still present in crowdfund and the attacker's address is the spender, they can easily move the NFT out of the contract.

Another assumption, that crowdfund would remain active until expiry if the contract doesn't own the NFT, won't hold true as well.

3. Transfer the NFT back to self or return it back to the marketplace by selling it on the collection floor.

ℹ️ If the attacker doesn't have initial funds to cover the buy cost, they can use the flash loan to buy NFT, given they have a buyer at the end to close the loop. If they don't have a specific buyer, they need to sell at the collection price; here, they would need to pay the difference. If they have a specific buyer — for example, some alternate party, competitor, or buyer from mempool — they can close it there.

ℹ️ If an attacker uses their own funds, they have the option to list NFT again rather than sell it. Here, though, you take risk of market volatility. So then the question is: How is it different from an attacker just buying NFT to grief crowdfund? In normal cases, the attacker needs to hold it for the whole duration to grief crowdfund. Whereas in an exploit case, you can list it soon and get it sold. So the time period over the risk that is taken is minimized.

Resolution

If NFT is considered gifted, consider directly sending it to an unreachable address like `address(0)`.

This way, all previous approvals would be canceled.

Technically, locking NFT inside the crowdfund contract forever, or sending it to a 0

address, is the same thing.

Below are a few other solutions that, though possible, are not ideal:

1. Assert `callTarget ≠ nftContract`: not ideal since this will block the action required for NFT initial mints.
2. After the external call, verify the spender: not possible to verify operators.

~~L-1~~ Someone can grief the creation of new crowdfunds by sending some ETH to the predicted address beforehand

TOPIC	STATUS	IMPACT	LIKELIHOOD
Griefing	Fixed	Low	Low

The logic now uses msg.value to assign the first contribution.

While creating crowdfund, if the deployer has passed some ETH, the intention is to credit `opts.initialContributor` with the initial contribution.

```
Crowdfund.sol L160

uint96 initialBalance = address(this).balance.safeCastUint256ToUint96(); /
if (initialBalance > 0) {
    // If this contract has ETH, either passed in during deployment or
    // pre-existing, credit it to the `initialContributor`.
    _contribute(opts.initialContributor, initialBalance, opts.initialDeleg
}
```

However, an external actor can force the code to take the path of `inititalBalance > 0`, even when the deployer did not pass any ETH, by predicting the future

contract address `function(factory's address, nonce)` and sending some ETH there beforehand.

By doing so, someone can grief the creation of "Crowdfunds with Zero initial contribution" as *`_mint`* will revert by `opts.initialContributor = address(0)`.

Consider checking for `msg.value` instead of `address(this).balance` or credit it to the `msg.sender`.

We understand that this has a very lower impact and likelihood, as the deployer can always send a new transaction with `intitalContributor ≠ address(0)`, but we wanted to report it because of its easy fix.

~~I-1~~ Voting power minted incorrectly for externally created parties would break various assumptions.

TOPIC	STATUS	IMPACT
Spec-breaking	Fixed	Informational ✳

Now the maximum minted voting power is capped to the totalVotingPower.

Party creators can mint tokens with equal or higher power than `totalVotingPower`, allowing them to propose and unanimously pass proposals instantly.

By doing this, any member owning an NFT with this power could bypass all the different protections for the precious NFTs by calling any proposal and having a unanimous vote because the votes are quantified with the `totalVotingPower`.

Party contracts with a total voting power higher than

`totalVotingPower` can leave members without a supply of Fractionalize ERC20 vault tokens.

Similar to the item above, the `Fractionalize` proposal type creates a distribution of governance tokens with a supply equal to the `totalVotingPower`, if there is more accumulated power than this value among the governance NFTs, possibly leaving contributors without any shares.

Party creators can mint tokens with `VETO_VALUE` voting power, disallowing the receiver to propose and vote.

The vetoed value in the proposal state info it's represented as is the `type(uint96).max` value; however, if a creator of a party assigns this value as a voting power for specific governance NFT, every attempt to propose would result in the proposal having the veto value. On the other hand, if this user tries to vote for any proposal, it will revert to an arithmetic error since the proposer will accept the proposal with their power, and the max value of votes (`type(uint96).max`) would be overflowed.

The vote quantity defined in `mint()` can overpass the `uint96` max value and revert to overflow on any `accept()` call.

The mint function takes the `votingPower` parameter as a `uin256` ; however, the votes are accounted for as a `uint96`, the same as the above item, and if an NFT governance token has more power than the `uint96` max value, the voting and proposals for that NFT would revert in every call.

Q-1 Lack of guarantee that the externally created party possesses the precious.

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Acknowledged	High

The `PartyDAO` specs suggest that a party could be created directly from the factory as well.

Since it's not coming from the crowdfunding route, the creator can pass anything as precious.

There is no verification if the party holds precious NFTs or not in party initialization.

At present, the creator of the party would need to transfer precious manually after the party initialization. Which they may choose not to do.

Consider transferring NFTs inside the party initialization to get rid of this trust assumption. We couldn't think of any impact this would have, except wrong information being given to the user and some added restrictions in arbitrary calls proposal — which is why we marked this as quality.

~~Q-2~~ Unsafe Multi-Signature Transfer

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	High

Now the multi-sig wallet update happens in two steps, one for assigning the new address, and the second one is accepting it.

In the Globals contract, consider making transferMultiSig() a two-step process, where the new multi-sig needs to accept the ownership to have an effect. This would protect from incorrect updates done on highly powerful functions.

Q-3 Non-standard ERC721s could cause odd behaviors

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Acknowledged	Medium

The party contracts assume that the underlying NFTs are following the ERC721 standard.

If any underlying NFTs are not following standard, the code could break at various points.

While doing `_setPreciousList` in party initialization, consider adding the `supportsInterface` check on passed NFTs.

~~Q-4~~ Lack of minimum cap for the cancellation delay

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Medium

There is a new global variable for the minimum cancel delay.

The Governance contracts have a `cancel()` function to prevent having a locked status on the party when a proposal has been in progress for an undesired amount of time. Although this function call can “unlock” the contract, it can also leave it in an unrecoverable state, even losing a precious asset.

To ensure that a cautious time is set to this inside the proposals, consider making

the minimum cap for the `cancelDelay` time the currently present cap for the `maximum` cancel delay.

Q-5 Accept function doesn't revert if the caller has zero voting power

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Acknowledged	Medium

In the PartyGovernance, the voting power used for the `accept()` function can be zero, and this can cause unnecessary gas spent if a user mistakenly calls the accept function without any voting power. If the likelihood of users calling accept even when they don't have voting power is high, consider adding a check on top.

~~Q-6~~ Redundant loop in `getContributorInfo()`

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Medium

The `ethOwed` + `ethUsed` is now being used if the party won.

For `getContributorInfo()` of `CrowdFund.sol`, there is no need to calculate `ethContributed` by looping over all contributions for won and lost states; the same result could be achieved by adding `ethOwed` and `ethUsed`.

~~Q-7~~ Inconsistent comments

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Medium

In the contract's code, there are a few inconsistencies with the spec and code logic in the comments regarding the voting power.

- In `PartyGovernance.sol`, line 226 and line 239:

```
// Caller must own a governance NFT at the current time.
    modifier onlyActiveMember() {
        ...
    }

// Caller must own a governance NFT at the current time or be the `Party`
    modifier onlyActiveMemberOrSelf() {
        ...
    }
```

- In `PartyGovernance.sol`, line 545:

```
/// @dev Only an active member (owns a governance token) can call this.
///      Afterwards, members can vote to support it with accept() or a par
///      host can unilaterally reject the proposal with veto().
```

- In `PartyGovernance.sol`, line 578:

```
/// @dev The voting power cast will be the effective voting power of the c
///      at the time propose() was called (see `getVotingPowerAt()`).
///      If the proposal reaches `passThresholdBps` acceptance ratio then
///      proposal will be in the `Passed` state and will be executable aft
```

```
///      the `executionDelay` has passed, putting it in the `Ready` state.
```

- In the PartyGovernance docs, in the **Voting Power Snapshots** section:

> When determining the effective voting power of a user, we binary search a user's voting power records for the most recent record <= the proposal time.

It should be `<` instead of `<=`.

~~Q-8~~ Imprecise Modifier name

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

Changed the function's logic to be executed by only hosts.

Consider renaming onlyHostCanBuy to a more appropriate name. The naming suggests that only the host can call buy; however, if set to true and there are gatekeeper addresses set, the code also allows gated contributors to buy.

Q-9 Double declaration of `Globals` address

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Acknowledged	Low

The `Globals` address variable is present in both the `CrowdFund` and `CrowdFundNFT` contracts as private immutable.

CrowdFund inherits from CrowdFundNFT, hence the same result could be achieved by having `Globals` as public or internal immutable inside the `CrowdFundNFT` contract only.

G-1 Unnecessary check for `_burn()` function

TOPIC	STATUS	GAS SAVINGS
Gas Optimization	Acknowledged	Medium

In the Crowdfund.sol contract, there is no need for the check at line 548 of the _burn() function. If Crowdfund state is won, its given party address will not be zero for all types of crowdfunding.

G-2 Redundant checks

TOPIC	STATUS	GAS SAVINGS
Gas Optimization	Acknowledged	Low

On line 105, `BuyCrowdfundBase` has the following check:

```
// Ensure the call target isn't trying to reenter
if (callTarget == address(this)) {
    revert InvalidCallTargetError(callTarget);
```

```
}
```

Although the `callTarget` will revert if someone tries to re-enter the contract, another smart contract could be the intermediary and achieve a re-entrant call, though it would be useless since there is another re-entrancy protection on line 127:

```
// Temporarily set to non-zero as a reentrancy guard.
settledPrice = type(uint96).max;
```

By setting the `settledPrice` to a value different from `0`, the function `buy()` is already guarded against any re-entrant call attempts because the function `getCrowdfundLifecycle` will make the execution revert to a `CrowdfundLifecycle.Busy` state.

The same thing is present in line 161, inside the `ArbitraryCallsProposal` contract:

```
// Cannot call ourselves.
if (call.target == address(this)) {
    return false;
}
```

And it is mitigated with a similar approach in the `ProposalExecutionEngine` contract, in line 162.

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Emergent team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.