MACIQF Security Review

Auditors
0xKaden, Security Researcher

June 25, 2024

1 Executive Summary

Over the course of 10 days in total, Allo engaged with 0xKaden to review MACIQF.

Metadata

Repository	Commit
MACIQF	5f8b6e3

Summary

Type of Project	Quadratic Funding
Timeline	June 3rd, 2024 - June 14th, 2024
Methods	Manual Review

Total Issues

Critical Risk	1
High Risk	5
Medium Risk	8
Low Risk	0
Informational	2
Gas Optimizations	6

Contents

2 Introduction

Allo is an open-source protocol that enables groups to efficiently and transparently allocate pooled capital. MACIQF is a capital-constrained, collusion-resistant quadratic funding strategy using MACI (Minimal Anti-Collusion Infrastructure).

The focus of the security review was on the following directories:

1. https://github.com/nijoe1/MACI_QF/tree/5f8b6e33d46c1f88314a7eae900fc2985f924d3c/contracts/contracts/ClonableMaciContracts
2. https://github.com/nijoe1/MACI_QF/tree/5f8b6e33d46c1f88314a7eae900fc2985f924d3c/contracts/contracts/strategies/qf-maci

Disclaimer: This review does not make any warranties or guarantees regarding the discovery of all vulnerabilities or issues within the audited smart contracts. The auditor shall not be liable for any damages, claims, or losses incurred from the use of the audited smart contracts.

3 Findings

3.1 Critical Risk

3.1.1 Anyones voice credit balance can be reused during `signUp`

Severity: Critical

Context: MACIQF.sol#L326

Description:

The flow for signing up a new public key with a voice credit balance is to go through `_allocate`, which performs necessary validation including checking proof of attendance and enforcing that the correct amount is transferred before assigning `contributorCredits` and signing up the user via `ClonableMACI.signUp` with the provided `signUpGatekeeperData` and `initialVoiceCreditProxyData`:

```
bytes memory signUpGatekeeperData = abi.encode(_sender, voiceCredits);
bytes memory initialVoiceCreditProxyData = abi.encode(_sender);

ClonableMACI(_maci).signUp(pubKey, signUpGatekeeperData, initialVoiceCreditProxyData);
```

Upon calling `ClonableMACI.signUp`, `register` is called to validate that the provided user address has a non-zero balance of `contributorCredits`, and if so, assigning those credits in the subsequent call to `getVoiceCredits`:

```
// Register the user via the sign-up gatekeeper. This function should
// throw if the user has already registered or if ineligible to do so.
signUpGatekeeper.register(msg.sender, _signUpGatekeeperData);

// Get the user's voice credit balance.
uint256 voiceCreditBalance = initialVoiceCreditProxy.getVoiceCredits(
    msg.sender,
    _initialVoiceCreditProxyData
);
```

The problem with this logic is that anyone can call `signUp` directly, providing arbitrary parameters and so long as the provided address has `contributorCredits`, a corresponding amount of voice credits will be applied to the provided `_pubKey`.

See below how `register` doesn't check the `_caller` and simply returns whether the provided `user` address has a non-zero amount of `contributorCredits` and how `getVoiceCredits` also doesn't check the `_caller` and simply returns the amount of `contributorCredits` the provided `_allocator` has:

```
function register(address /* _caller */, bytes memory _data) external view {
    if (msg.sender != _maci) {
        revert OnlyMaciCanRegisterVoters();
    }

    address user = abi.decode(_data, (address));
    bool verified = contributorCredits[user] > 0;

    if (!verified) {
        revert UserNotVerified();
    }
}
```

```
function getVoiceCredits(
    address /* _caller */,
    bytes memory _data
) external view returns (uint256) {
    address _allocator = abi.decode(_data, (address));
    if (!_isValidAllocator(_allocator)) {
        return 0;
    }
    return contributorCredits[_allocator];
}
```

The result of this is that any attacker can call `signUp`, passing the address of the user with the largest `contributorCredits` balance to end up with the same amount of voice credits, allowing them to manipulate the result of the poll.

Recommendation:

Enforce that the `_caller` parameter provided to `register` and `getVoiceCredits` is `address(this)`

to validate that the execution started in `_allocation`, which contains necessary validation. Furthermore, since the call to `register` in `signUp` expects execution to revert if the user has already signed up, implement a storage mapping to keep track of signed up contributors and revert if a user has already signed up in `register`.

Allo: Fixed in c70bdfd by reverting in case the user has already signed up.

Kaden: While this does appear to fix the issue, I would recommend also validating that the `signUp` was initiated in `MACIQF._allocate` by validating that the `_caller == address(this)` in `register` and `getVoiceCredits` to prevent any unexpected behavior.

Allo: Fixed in 4377689.

Kaden: Fixed.

3.2 High Risk

3.2.1 Poll DoS possible by publishing a message with an `_encPubKey.y = 1`

Severity: High

Context: `ClonablePoll.sol#L210`

Description:

In `ClonablePoll.publishMessage`, the user provides an `_encPubKey` parameter that is validated to have `x` and `y` coordinates less than the `SNARK_SCALAR_FIELD` constant, which is intended to prove that it is a valid public key:

```
// validate that the public key is valid
if (_encPubKey.x >= SNARK_SCALAR_FIELD || _encPubKey.y >= SNARK_SCALAR_FIELD) {
    revert MaciPubKeyLargerThanSnarkFieldSize();
}
```

However, as was noted in a recent post-mortem, the above validation doesn't actually validate that the provided `_encPubKey` is a point on the Baby JubJub elliptic curve used by the protocol.

As a result, there is a DoS vector wherein an attacker can provide an `_encPubKey` with a `y` coordinate of 1, which ultimately leads to a divide by zero error in the zk-SNARK circuit which the public key is passed to. This prevents proof generation for message processing which prevents the poll from ever being completed. See the post-mortem for a full explanation.

Recommendation:

This issue can be fixed by validating that the `_encPubKey` is a valid point on the Baby JubJub elliptic curve. This can be checked with the following function:

```
/**
 * @dev Check if a given point is on the curve
 * (168700x^2 + y^2) - (1 + 168696x^2y^2) == 0
 */
function isOnCurve(uint256 _x, uint256 _y) internal pure returns (bool) {
  uint256 xSq = mulmod(_x, _x, Q);
  uint256 ySq = mulmod(_y, _y, Q);
  uint256 lhs = addmod(mulmod(A, xSq, Q), ySq, Q);
  uint256 rhs = addmod(1, mulmod(mulmod(D, xSq, Q), ySq, Q), Q);
  return submod(lhs, rhs, Q) == 0;
}
```

Which can be implemented with the following:

```
if (!CurveBabyJubJub.isOnCurve(_encPubKey.x, _encPubKey.y)) {
  revert InvalidPubKey();
}
```

Note that there are several times in the codebase in which a public key is provided which should be validated to be a point on the Baby JubJub curve. See MACI v1.2.3 implementations to see where this validation is necessary.

Allo: Fixed in ecfa260 and e1fc0c5 by validating that provided public keys are on the Baby JubJub elliptic curve.

Kaden: Fixed.

3.2.2 Matching pool tokens get locked in the contract if the pool manager isn't the first to withdraw after cancellation

Severity: High

Context: `MACIQFBase.sol#L346`

Description:

In case the poll gets cancelled, the pool manager can withdraw matching pool tokens via `MACIQFBase.withdraw`:

```
// Only if the pool is cancelled the funds can be withdrawn
// Otherwise the funds will be taken from the winners of the pool
if (!isCancelled) {
    revert INVALID();
}
// Transfer only the amount used in the matching pool and not the total balance
// Which includes the contributions. This is to ensure if the round is cancelled
// Contributors can withdraw their contributions.
uint256 amount = _getBalance(_token, address(this)) - totalContributed;
_transferAmount(_token, msg.sender, amount);
```

This logic asserts that the matching pool amount can always be computed as the contract balance of the token minus `totalContributed`. However, the problem with this logic is that `totalContributed` stays fixed while contributors can withdraw their contribution via `withdrawContributions`, reducing the contract balance of the token:

```
address contributor = _contributors[i];
uint256 amount = contributorCredits[contributor] * voiceCreditFactor;
if (amount > 0) {
    // Reset before sending funds the contributor credits to prevent Re-entrancy
    contributorCredits[contributor] = 0;
    if (allo.getPool(poolId).token != NATIVE) {
        _transferAmountFrom(
            allo.getPool(poolId).token,
            TransferData(address(this), contributor, amount)
        );
    } else {
        _transferAmountFrom(NATIVE, TransferData(address(this), contributor, amount));
    }
    result[i] = true;
} else {
    result[i] = false;
}
```

As a result, if contributors withdraw their contribution prior to the pool manager withdrawing the matching pool, some amount of matching pool tokens will be permanently locked in the contract.

Recommendation:

To prevent this, it's necessary to decrement `totalContributed` by the amount of tokens withdrawn by contributors via `withdrawContributions`, e.g.:

```
address contributor = _contributors[i];
uint256 amount = contributorCredits[contributor] * voiceCreditFactor;
if (amount > 0) {
+  totalContributed -= amount;
```

Allo: Fixed in 6487c23.

Kaden: Fixed.

3.2.3 Attackers can use other users' proof of attendance by frontrunning

Severity: High

Context: ZuPassRegistry.sol#L101

Description:

In `MACIQF._allocate`, the provided `_data` is decoded and includes proof data for Zupass proof of attendance which is validated by passing it to `ZupassRegistry.validateProofOfAttendance`:

```
(
    PubKey memory pubKey,
    uint256 amount,
    uint[2] memory _pA,
    uint[2][2] memory _pB,
    uint[2] memory _pC,
    uint[38] memory _pubSignals
) = abi.decode(_data, (PubKey, uint256, uint[2], uint[2][2], uint[2], uint[38]));

...

// Validate proof of attendance if provided
if (_pA[0] != 0) {
    if (!zupassVerifier.validateProofOfAttendance(_pA, _pB, _pC, _pubSignals)) {
        revert InvalidProof();
    }
    if (amount > maxContributionAmountForZupass) {
        revert ContributionAmountTooLarge();
    }
```

Providing a valid proof allows the user to contribute a greater amount, providing them a greater voting power for quadratic funding.

The problem with this logic is that the provided proof data is never validated to be associated with the caller or the provided `_pubKey`. This allows an attacker to frontrun allocation transactions with a provided proof of attendance by using the same proof to receive a higher than intended voting power, allowing them to manipulate the poll results and preventing the user that initially provided the proof from using their own valid proof.

Recommendation:

Include the expected sender in the proof and revert if the proof cannot be verified or if the `_sender` is not the expected sender.

Allo: Fixed in 28a27fc.

Kaden: The provided fix uses `tx.origin` for authorization which should be avoided for two reasons:

- If an attacker can get a victim to call a contract they control for any reason then this attack can still be executed as `tx.origin` will still be as expected.
- It also prevents users of smart contract wallets from participating since the `tx.origin` will be the EOA that called the smart contract wallet and not the smart contract wallet itself.

Instead, we should validate against `msg.sender` in the `MACIQF` contract in the case that a valid proof was provided.

Allo: Fixed in 4377689.

Kaden: While this fixes the original issue, the fix introduces a new bug where it allows for invalid proofs to be accepted by not reverting if `watermark == 0`, which is returned for an invalid proof.

Allo: Fixed in 829eac5.

Kaden: Fixed.

3.2.4 Sybil attacks

Severity: High

Context: MACIQF.sol#L242

Description:

In any quadratic funding mechanism, since allocated funds are matched, if it's possible to allocate funds from any account then it's possible to sybil attack the system. A sybil attack is when an attacker executes some logic from many different accounts providing some increased ability over executing the same logic from one account, e.g. an attacker provides an amount of 10 from 5 different accounts and receives more total voting power than if they were to provide an amount of 50 from a single account.

Sybil attacks are a risk in quadratic funding mechanisms because they implicitly rely on the expectation that each participant is unique by providing more weight to the uniqueness of voters than the amount of funds they're providing. Since `MACIQF` allows any account to allocate funds, it fails to prevent against sybil attacks.

Recommendation:

The only way to prevent sybil attacks is to somehow enforce that each user that signs up is unique, e.g. using one of the following:

- Proof of attendance
- KYC
- Allowlist

Allo: Acknowledged and managed operationally by setting the contribution limit for non-allowlisted members to 0 or some really small value.

Kaden: Acknowledged.

3.2.5 Funds are not distributed to intended `recipientAddress`

Severity: High

Context: MACIQF.sol#L314

Description:

When recipients register, they provide a `recipientAddress` to indicate where the funds should be distributed to:

```
recipient.recipientAddress = recipientAddress;
```

However, in `_distributeFunds`, we actually distribute funds to the `recipientId` address instead of the `recipientAddress`:

```
_transferAmount(pool.token, recipientId, amount);
```

As a result, the distributed funds could be inaccessible, e.g. if the recipient is registered via a contract that is not intended to receive tokens.

Recommendation:

Distribute tokens to the `recipientAddress` instead of the `recipientId`:

```
-_transferAmount(pool.token, recipientId, amount);
+_transferAmount(pool.token, recipient.recipientAddress, amount);
```

Allo: Fixed in 0a28461.

Kaden: Fixed.

3.3 Medium Risk

3.3.1 `voiceCreditFactor` is incorrectly set if a non-native token is used

Severity: Medium

Context: MACIQFBase.sol#L236

Description:

In `__MACIQFBaseStrategy_init`, we set the `voiceCreditFactor` according to the following:

```
// Calculate the voice credit factor
voiceCreditFactor = (MAX_CONTRIBUTION_AMOUNT * tokenDecimals) / MAX_VOICE_CREDITS;
voiceCreditFactor = voiceCreditFactor > 0 ? voiceCreditFactor : 1;
```

Prior to doing this, we retrieve `tokenDecimals`:

```
uint256 tokenDecimals;
if (address(pool.token) == NATIVE) {
    tokenDecimals = 10 ** 18;
} else {
    tokenDecimals = ERC20(pool.token).decimals();
}
```

The problem is that in retrieving the `tokenDecimals` value, for native tokens we set it as 10^18 while for non-native tokens, we just set it as the number of decimals that the token has, e.g. 18. The result of this is that the `voiceCreditFactor` is set many orders of magnitude lower for non-native tokens than intended.

Recommendation:

As is done for the native token, compute `tokenDecimals` as 10^`decimals()` for non-native tokens as well:

```
uint256 tokenDecimals;
if (address(pool.token) == NATIVE) {
    tokenDecimals = 10 ** 18;
} else {
    tokenDecimals = 10 ** ERC20(pool.token).decimals();
}
```

Allo: Fixed in 729dda6.

Kaden: Fixed.

3.3.2 Infinite loop in `reviewRecipients` when a recipient without a status is provided

Severity: Medium

Context: MACIQFBase.sol#L296-L298

Description:

In `reviewRecipients`, within the `for` loop, if a provided recipient has a `status` of `Status.None`, we `continue` to the next iteration of the loop:

```
// If the recipient is not in review, skip the recipient
// This is to prevent updating the status of a recipient that is not registered
if (recipient.status == Status.None) {
    continue;
}
```

The problem with this logic is that we don't actually increment `i` until the end of the loop iteration, so when we `continue`, we repeat the same iteration, which will cause us to loop until the transaction runs out of gas.

Recommendation:

Immediately prior to the `continue`, increment `i`:

```
if (recipient.status == Status.None) {
+   unchecked {
+       i++;
+   }
    continue;
}
```

Allo: Fixed in 20f5cb7.

Kaden: Fixed.

3.3.3 Users can fund the pool without paying the Allo fee

Severity: Medium

Context: MACIQF.sol#L492

Description:

Normally when funding pools, we must do so via `Allo._fundPool`, which causes the `poolAmount` to be incremented accordingly, taking a fee in the process:

```
if (percentFee > 0) {
    feeAmount = (_amount * percentFee) / getFeeDenominator();
    amountAfterFee -= feeAmount;

    if (feeAmount + amountAfterFee != _amount) revert INVALID();

    if (_token == NATIVE) {
        _transferAmountFrom(
            _token,
            TransferData({from: msg.sender, to: treasury, amount: feeAmount})
        );
    } else {
        uint256 balanceBeforeFee = _getBalance(_token, treasury);
        _transferAmountFrom(
            _token,
            TransferData({from: msg.sender, to: treasury, amount: feeAmount})
        );
        uint256 balanceAfterFee = _getBalance(_token, treasury);
        // Track actual fee paid to account for fee on ERC20 token transfers
        feeAmount = balanceAfterFee - balanceBeforeFee;
    }
}

if (_token == NATIVE) {
    _transferAmountFrom(
        _token,
        TransferData({from: msg.sender, to: address(_strategy), amount:
        ↪  amountAfterFee})
    );
} else {
    uint256 balanceBeforeFundingPool = _getBalance(_token, address(_strategy));
    _transferAmountFrom(
        _token,
        TransferData({from: msg.sender, to: address(_strategy), amount:
        ↪  amountAfterFee})
    );
    uint256 balanceAfterFundingPool = _getBalance(_token, address(_strategy));
    // Track actual fee paid to account for fee on ERC20 token transfers
    amountAfterFee = balanceAfterFundingPool - balanceBeforeFundingPool;
}

_strategy.increasePoolAmount(amountAfterFee);
```

In `MACIQF`, however, we don't use the `poolAmount`, rather we just use the token balance of the

contract directly:

```
uint256 _poolAmount = _getBalance(allo.getPool(poolId).token, address(this));
```

As a result, it's possible to fund the pool by sending tokens directly to the contract to avoid paying the Allo fee.

Recommendation:

Instead of using the token balance of the contract directly, use the `poolAmount` storage variable directly to ensure that funding can only be provided by paying the Allo fee:

```
// finalize()
-uint256 _poolAmount = _getBalance(allo.getPool(poolId).token, address(this));
-alpha = calcAlpha(_poolAmount, totalVotesSquares, _totalSpent);
+alpha = calcAlpha(poolAmount, totalVotesSquares, _totalSpent);
matchingPoolSize = _poolAmount - _totalSpent * voiceCreditFactor;
```

Allo: Fixed in a667f3c.

Kaden: Fixed.

3.3.4 Logical error prevents allocation with non-native tokens

Severity: Medium

Context: MACIQF.sol#L230

Description:

In `MACIQF._allocate`, we enforce that the `amount == msg.value` or else we revert:

```
if (amount != msg.value) revert INVALID();
```

However, if the selected pool token is non-native then we shouldn't have to transfer native tokens.

We validate later on that the required amount, whether native or not is provided:

```
if (token != NATIVE) {
    _transferAmountFrom(token, TransferData(_sender, address(this), amount));
} else {
    if (msg.value != amount) revert InvalidAmount();
}
```

As a result, it's only possible to allocate tokens to a `MACIQF` pool by also transferring the same amount of native tokens, which is unintended.

Recommendation:

Remove the initial `msg.value` check:

```
if (contributorCredits[_sender] != 0) revert AlreadyContributed();
-if (amount != msg.value) revert INVALID();
if (amount > MAX_VOICE_CREDITS * voiceCreditFactor) revert
↪  ContributionAmountTooLarge();
```

Allo: Fixed in 45d444d.

Kaden: Fixed.

3.3.5 Recipients can change their data after being accepted

Severity: Medium

Context:

- MACIQFBase.sol#L453
- MACIQFBase.sol#L283

Description:

In `_registerRecipients`, recipients provide data for review by the pool manager. Recipients can update this data at any time in the registration period by calling the function again with new data. Even if the recipient has already been accepted, they can still go back and re-register to change their data, while maintaining the accepted status.

As a result, it's possible for recipients to change their data after being accepted by the pool manager, effectively causing the pool manager to unexpectedly accept a recipient that is not aligned with that which they intended to accept.

Recommendation:

In `_registerRecipients`, if a recipient has previously been accepted and they change their data, their status should be changed to `Pending` or `InReview` to better reflect their change in status.

Furthermore, we also need to prevent the recipient from being able to frontrun reviews by changing their status. This can be prevented operationally by only allowing `reviewRecipients` to be called after the registration period is complete, preventing overlap between these two functions. Alternatively, we can prevent this by including a hash of the expected state of each recipient to be verified against the current state of the recipient.

Allo: Fixed in 2418e92 and b5ebd76.

Kaden: The provided fix prevents recipients from changing their data after being accepted, however, it does not prevent users from changing their data via frontrunning `reviewRecipients`. Pausing registration may be helpful in preventing frontrunning attacks but does not necessarily solve the problem. Instead, the only way to fully prevent this frontrunning attack is to provide a hash of the expected data in `reviewRecipients` or to validate recipients against a timestamp which is updated every time the user registers.

Allo: Fixed in 829eac5.

Kaden: Fixed.

3.3.6 Centralization risks

Severity: Medium

Context:

- ClonableTally.sol#L146
- MACIQF.sol#L456

Description:

Both the `coordinator` and the `poolManager` are permissioned actors with centralized control over important aspects of the protocol:

- The `coordinator` has the ability to prevent the `MACI` tally from ever being completed since they are the only entity capable of tallying the results.
- The `poolManager` can prevent the round from being finalized by simply not calling `finalize`.
- If the `coordinator` and `poolManager` collude, they can prevent a round from being tallied or finalized and then also not cancel the round, causing all funds to be unwithdrawable.

Recommendation:

Ensure there is clear, user-facing documentation which indicates the centralization risks associated with interacting with these contracts.

Allo: Partially fixed in e34ca1f by only allowing the `coordinator` to call `finalize` and will provide user-facing documentation.

Kaden: Centralization risks are still present but the issue has been mitigated as recommended.

3.3.7 Previously accepted removed recipients can still be voted for, causing a loss of funds

Severity: Medium

Context: MACIQFBase.sol#L283

Description:

When a recipient is accepted, their `recipientId` is indexed with a vote index that can be voted for on the `ClonablePoll` contract. However, if they're later removed, they can still be voted for. This will affect accounting related to the total amount of votes, but the recipient cannot be distributed to since they are no longer accepted, thereby causing some amount of funds to be permanently locked in the contract.

This issue is exacerbated by the fact that the voting, registration, and review periods all overlap, leading to a possible case in which someone votes for an accepted recipient which is later removed.

Recommendation:

As noted in 3.3.5, we can prevent the latter case by only allowing `reviewRecipients` to be called after the registration period. While we can't entirely prevent voters from voting for removed recipients, clear, user-facing documentation should be added to indicate that voters should carefully validate that the provided vote index corresponds to an accepted recipient prior to voting.

Allo: Fixed in 2418e92.

Kaden: Fixed.

3.3.8 If the same recipient has multiple tally results or is not accepted, the total vote accounting will increase unexpectedly

Severity: Medium

Context: MACIQF.sol#L421-L423

Description:

Each `recipient` is only expected to have one tally result, but in the case that they have more than one tally result, each additional result will be used to increment total vote accounting (`totalRecipientVotes` & `totalVotesSquares`), but will not increase the recipients vote accounting (`recipient.totalVotesReceived`):

```
totalRecipientVotes += _tallyResult;
totalVotesSquares = totalVotesSquares + (_tallyResult * _tallyResult);

...

if (recipient.tallyVerified) {
    return;
}
```

The result of this is that the additional result votes will be unused and thus a corresponding share of the tokens will be permanently locked in the contract.

The same issue applies to the validation that the recipient is an accepted recipient:

```
if (!_isAcceptedRecipient(recipientId)) return;
```

Recommendation:

Check whether the recipient has already had a tally result applied or if they're not an accepted recipient prior to modifying total vote accounting data and return at that point instead.

Allo: Fixed in 2418e92.

Kaden: Fixed.

3.4 Low Risk

No low risk findings were discovered.

3.5 Informational

3.5.1 Hardcoded `emptyBallotRoots` doesn't allow for different state tree depths

Severity: Informational

Context: `ClonablePoll.sol#L322`

Description:

In `ClonablePoll`, the `emptyBallotRoots` are hardcoded:

```
function _setEmptyBallotRoots() internal {
    emptyBallotRoots[0] = uint256(
        4904028317433377177773123885584230878115556059208431880161186712332781831975
    );
    emptyBallotRoots[1] = uint256(
        3447323123500529440411043453252951114087479753389084917638178720571388641 63
    );
    emptyBallotRoots[2] = uint256(
        1944581445501297879948389281195039638308418321086027992320717668249048990 7069
    );
    emptyBallotRoots[3] = uint256(
        1062181078069030348282742214338985804982967022224490061765240467212549201 3328
    );
    emptyBallotRoots[4] = uint256(
        1707769037933702617943804460206808569066204346464351154432965614099739049 8741
    );
}
```

The `emptyBallotRoots` depends on the state tree depth and as a result, by hardcoding the `emptyBallotRoots`, only one state tree depth can be used for all instances.

Recommendation:

Allow for different `emptyBallotRoots` to be used during deployment of the `ClonablePoll` contract via the `ClonableMACIFactory`.

Allo: Fixed in 0364b47.

Kaden: Fixes the primary issue but assumes that the `_emptyBallotRoots` will always have a length of 5. In case it may not, we should instead loop to `_emptyBallotRoots.length`.

Allo: Fixed in 4377689.

Kaden: Fixed.

3.5.2 Internal functions should be consistently prefixed with an underscore

Severity: Informational

Context:

- MACIQF.sol#L501
- MACIQFBase.sol#L616

Description:

It's best practice to prefix the names of internal functions with an underscore to clearly indicate that they are internal. There are a couple instances where internal functions in the codebase are not prefixed with an underscore: `verifyClaim` and `getAllocatedAmount`, as linked above.

Recommendation:

Prefix these function names with underscores.

Allo: Fixed in 4377689.

Kaden: Fixed.

3.6 Gas Optimizations

3.6.1 Redundant strategy retrieval

Severity: Gas optimization

Context: MACIQF.sol#L189

Description:

`__MACIQFStrategy_init` contains logic to retrieve the pool `strategy`:

```
address strategy = address(allo.getPool(_poolId).strategy);
```

However, the `strategy` address is simply the current contract, i.e. `address(this)`.

Recommendation:

Remove the logic to retrieve `strategy` and instead replace it's usage in the function with `address(this)`:

```
-address strategy = address(allo.getPool(_poolId).strategy);

...

-_maci = _maciFactory.createMACI(strategy, strategy, coordinator,
↪  _params.maciParams.maciId);
+_maci = _maciFactory.createMACI(address(this), address(this), coordinator,
↪  _params.maciParams.maciId);
```

Allo: Fixed in a4fea31.

Kaden: Fixed.

3.6.2 Excessive gas costs for `_allocate` calls without proof of attendance

Severity: Gas optimization

Context: MACIQF.sol#L221-L225

Description:

In `_allocate`, full proof arrays are required to be provided in the case that a proof of attendance is provided:

```
(
    PubKey memory pubKey,
    uint256 amount,
    uint[2] memory _pA,
    uint[2][2] memory _pB,
    uint[2] memory _pC,
    uint[38] memory _pubSignals
) = abi.decode(_data, (PubKey, uint256, uint[2], uint[2][2], uint[2], uint[38]));

...

// Validate proof of attendance if provided
if (_pA[0] != 0) {
    if (!zupassVerifier.validateProofOfAttendance(_pA, _pB, _pC, _pubSignals)) {
        revert InvalidProof();
    }
```

However, even if proof of attendance is not provided, `abi.decode` requires that entire proof arrays are provided since it will revert if it cannot decode the expected types. This means that multiple empty arrays must be provided as calldata even though they will not be used at all, costing a significant amount of unnecessary gas.

Recommendation:

Provide a boolean as part of the encoded `_data` to indicate whether proof of attendance is being supplied with the call and only decode the proof of attendance if `true`:

```
(
    PubKey memory pubKey,
    uint256 amount,
    bool proofOfAttendance
) = abi.decode(_data, (PubKey, uint256, bool));

if (proofOfAttendance) {
    (
        ,
        ,
        ,
        uint[2] memory _pA,
        uint[2][2] memory _pB,
        uint[2] memory _pC,
        uint[38] memory _pubSignals
    ) = abi.decode(_data, (PubKey, uint256, bool, uint[2], uint[2][2], uint[2],
    ↪  uint[38]));
}
```

We can then check whether we need to call `zupassVerified.validateProofOfAttendance` based on whether `proofOfAttendance` is `true`:

```
// Validate proof of attendance if provided
if (proofOfAttendance) {
    if (!zupassVerifier.validateProofOfAttendance(_pA, _pB, _pC, _pubSignals)) {
        revert InvalidProof();
    }
```

Note: The above examples have not been tested.

Allo: Fixed in 6b0a452.

Kaden: Fixed.

3.6.3 Redundant native token check

Severity: Gas optimization

Context: MACIQF.sol#L570-L577

Description:

In `withdrawContributions`, when transferring funds to the contributor, we have an `if/else` statement to use different logic if the pool token is native:

```
if (allo.getPool(poolId).token != NATIVE) {
    _transferAmountFrom(
        allo.getPool(poolId).token,
        TransferData(address(this), contributor, amount)
    );
} else {
    _transferAmountFrom(NATIVE, TransferData(address(this), contributor, amount));
}
```

However, both cases are effectively identical since if `allo.getPool(poolId).token == NATIVE`, we provide `NATIVE` as the token to transfer, which would be the same as providing `allo.getPool(poolId).token`.

Recommendation:

Remove the `if/else` statement and simply transfer the pool tokens as usual:

```
_transferAmountFrom(
    allo.getPool(poolId).token,
    TransferData(address(this), contributor, amount)
);
```

Allo: Fixed in 7c42ffe.

Kaden: Fixed.

3.6.4 Redundant timestamp checks

Severity: Gas optimization

Context: MACIQFBase.sol#L369

Description:

`_isPoolTimestampValid` provides timestamp validation given the start and end time of the registration and allocation periods, reverting if the start and end time of each period is not in order or if the allocation period is not strictly after the registration period:

```
if (
    _registrationStartTime > _registrationEndTime ||
    _registrationStartTime > _allocationStartTime ||
    _registrationEndTime > _allocationEndTime ||
    _allocationStartTime > _allocationEndTime ||
    // Added condition to ensure registrationEndTime cannot be greater than
    ↪  allocationStartTime
    // This is to prevent accepting a recipient after the allocation has started
    // Because in MACI votes are encrypted if a recipient is REJECTED after the
    ↪  allocation has started
    // the votes for that recipient will be wasted toghether with the matching funds of
    ↪  the contributors
    _registrationEndTime > _allocationStartTime
) {
    revert INVALID();
}
```

However, we can simplify this logic to remove a couple cases.

Since we know that `_registrationStartTime <= _registrationEndTime`, `_allocationStartTime <= _allocationEndTime` and `_registrationEndTime <= _allocationStartTime`, it must also be true that `_registrationStartTime <= _allocationStartTime` and `_registrationEndTime <= _allocationEndTime`. As such, we can remove these two checks.

Recommendation:

Remove the redundant checks:

```
if (
    _registrationStartTime > _registrationEndTime ||
-   _registrationStartTime > _allocationStartTime ||
-   _registrationEndTime > _allocationEndTime ||
    _allocationStartTime > _allocationEndTime ||
    // Added condition to ensure registrationEndTime cannot be greater than
    ↪  allocationStartTime
    // This is to prevent accepting a recipient after the allocation has started
    // Because in MACI votes are encrypted if a recipient is REJECTED after the
    ↪  allocation has started
    // the votes for that recipient will be wasted together with the matching funds of
    ↪  the contributors
    _registrationEndTime > _allocationStartTime
) {
    revert INVALID();
}
```

Allo: Fixed in 9025795.

Kaden: Fixed.

3.6.5 Redundant `_isValidAllocator` check in `getVoiceCredits`

Severity: Gas optimization

Context: MACIQFBase.sol#L547-L549

Description:

In `getVoiceCredits`, prior to retrieving the amount of `contributorCredits` that the provided `_allocator` has, we check `_isValidAllocator` and return 0 if the `_allocator` is not valid:

```
address _allocator = abi.decode(_data, (address));
if (!_isValidAllocator(_allocator)) {
    return 0;
}
return contributorCredits[_allocator];
```

However, `_isValidAllocator` simply returns whether or not the `_allocator` has a non-zero amount of `contributorCredits`:

```
return contributorCredits[_allocator] > 0;
```

As a result, if we remove the `_isValidAllocator` check, we will end up with the same result since if the `_allocator` has 0 `contributorCredits`, we will just return 0 regardless.

Recommendation:

Remove the `_isValidAllocator` check:

```
address _allocator = abi.decode(_data, (address));
-if (!_isValidAllocator(_allocator)) {
-   return 0;
-}
return contributorCredits[_allocator];
```

Allo: Fixed in feb7658.

Kaden: Fixed.

3.6.6 Use events instead of storing data in `recipientToVoteIndex` mapping

Severity: Gas optimization

Context: MACIQFBase.sol#L171

Description:

`recipientToVoteIndex` is a storage mapping which is solely used to store the vote index for a given recipient:

```
recipientToVoteIndex[recipientId] = acceptedRecipientsCounter;
```

Since we never reference this data on-chain, it doesn't need to be stored on-chain. Instead, we can emit an event when a value would otherwise be stored in the mapping, which we can then index and store off-chain.

Recommendation:

Remove the `recipientToVoteIndex` mapping and replace its usage with an emitted event which is indexed off-chain.

Allo: Fixed in b46169b.

Kaden: Fixed.