Perfect Abstractions

Pixel Vault NFT Governance Audit

Table of Contents

1 Pixel Vault NFT Governance Audit

Perfect Abstractions conducted a smart contract audit of Pixel Vault's NFT Governance Contracts from September 15th to September 29th, 2022.

The git commit hash used for the audit is `eb2fb0f8c1bad2ca6a0ecb344994c3004205feee`.

Auditors:

- Zac Denham

Audit report reviewed by Nick Mudge.

1.1 Overview

The Pixel Vault NFT governance contracts enable any ERC721 NFT to be used as an opt-in governance token. This could allow for new forms of utility and coordination amongst NFT holders.

It works by implementing a wrapper token called the "Community Token." End users can lock their NFT in the contract in exchange for a community token that corresponds to one governance vote. This vote can be delegated to another address for representation. Community tokens are not tradable on secondary markets (transfers are disabled).

The CommunityToken code makes use of inheritance to modularize functionality. It is broken up into the following main contracts, with each inheriting the previous:

- `CommunityToken` which is based on ERC721
- `ERC721Wrapper` which handles receiving and disbursing the root token NFT in exchange for community tokens
- `ERC721WrapperVotes` which adds checkpointing for historical vote balances as well as delegation functionality
- `PVFDWrapperVotes` which adds `tokenUri` functionality

The governance itself is based on Compound's popular "Governor Bravo" contract. More detailed documentation on compound governance can be found here.

One notable governance gas optimization is that not all "Ballots" are stored on chain. Only the vote totals are accounted for, and whether a user has voted on a given proposal is packed into one bit at a position that corresponds to the user's "community id". This optimization reduces the number of "cold" storage slots that need to be updated in order to vote, in the happy case only requiring one slot to be updated. This effectively decreases the gas cost to vote, which might in turn incentivize voter participation.

The quorum mechanic also differs from Compound in that it is dynamically calculated so as to account for fluctuating supply of community tokens vs. Compound which uses a fixed immutable number.

Similar to the community token, governance uses inheritance:

- `Governor` - based on Compound governance, handles the proposal and voting lifecycle
- `PFVDGovernor` - adds dynamic quorum functionality based on community token totalSupply

This audit is an assessment of the Pixel Vault NFT governance system.

1.1.1 Objectives

1. Find bugs, inefficiencies and security vulnerabilities in the code base.
2. Make recommendations concerning bugs, inefficiencies and security vulnerabilities found as well as other recommendations that may improve the code base.

1.1.2 Scope

The following files were audited:

- contracts/CommunityToken.sol
- contracts/ERC721Wrapper.sol
- contracts/ERC721WrapperVotes.sol
- contracts/Governor.sol
- contracts/GovernorEvents.sol
- contracts/GovernorTypes.sol
- contracts/PVFDGovernor.sol
- contracts/PVFDWrapperVotes.sol
- contracts/TimeCheckpoint.sol
- contracts/TimelockEvents.sol
- contracts/TokenReceiver.sol

I. Medium Risk

2 Root Token Security

Medium Risk

It is worth noting that the wrapper token and governance integrity are highly dependent on the `rootToken` being benevolent and bug-free. **If a rootToken contract or owner account is compromised or otherwise malicious, critical governance mechanisms can be bypassed.**

For instance, say a malicious root token owner dev mints new tokens (e.g. the BAYC contract has this ability). They can then accumulate votes and pass proposals to drain treasuries dictated by the governance mechanism.

Similarly, if the root token contract is malicious, it might make arbitrary calls to `onERC721Received` in ERC721Wrapper.sol so as to mint additional voting units to a given address.

```
function onERC721Received(
    address, /* operator */
    address from,
    uint256 tokenId,
    bytes calldata /* data */
) external virtual override returns (bytes4) {
    require(
        msg.sender == address(rootToken),
        "ERC721Wrapper::onERC721Received: NFT not root NFT"
    );
    _mint(from, tokenId);
    _onTokenWrap(from, 1);

    return IERC721Receiver.onERC721Received.selector;
}
```

2.1 Recommendation

This list is not exhaustive:

- Before implementing PVFDWrapperVotes / PVFDGovernance with a given root token, the token should be thoroughly audited and investigated. Preferably it should also be immutable and with weak administrative capabilities.
- Although it is not in the base standard, many ERC721 tokens implement an immutable `totalSupply` function, you could limit the rootTokens to those that implement this, and use it to better validate against supply manipulation.
- Ensure there is not any admin functionality around burning or otherwise transferring root token NFTs, this can lead to exploits in the wrapper governance.
- Ensure there are no external calls in root token "transfer hooks"

3 Signature Replay Vulnerability

⚠ **Medium Risk**

3.1 Vulnerability

In ERC721Wrapper.sol `wrapBySig` and `unwrapBySig` functions, a nonce is passed in both function arguments, but the nonce's uniqueness is never enforced in the function logic. This leaves these functions open to signature replay attacks in which the original signature is used again at a later time against end user wishes (given the signature expiry is high enough).

3.1.1 Code

```
function wrapBySig(
    uint256 tokenId,
    uint256 nonce, // passed here
    uint256 expiry,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    require(
        block.timestamp <= expiry,
        "ERC721Wrapper::wrapBySig: signature expired"
    );

    // included in the struct hash
    bytes32 structHash = keccak256(
        abi.encode(_WRAP_TYPEHASH, tokenId, nonce, expiry)
    );
    bytes32 digest = keccak256(
        abi.encodePacked("\x19\x01", _domainSeparator(), structHash)
    );
    address signer = ecrecover(digest, v, r, s);

    require(
        signer != address(0),
        "ERC721Wrapper::wrapBySig: invalid signature"
    );

    // Never verified

    _wrap(signer, tokenId);
    _onTokenWrap(signer, 1);
}
```

3.2 Test Case

```
it('is able to be replayed (test passing is undesirable)', async function () {
  await testNFT.mint(accounts[0].address, 1);
  await testNFT.mint(accounts[0].address, 2);
  await testNFT.setApprovalForAll(erc721WrapperVotes.address, true);

  const { chainId } = await ethers.provider.getNetwork();
  const sig0 = await accounts[0]._signTypedData(
    {
      name: 'Wrapped Test Token',
      chainId,
      verifyingContract: erc721WrapperVotes.address,
    },
    {
      Wrap: [
        { name: 'tokenId', type: 'uint256' },
        { name: 'nonce', type: 'uint256' },
        { name: 'expiry', type: 'uint256' },
      ],
    },
    { tokenId: 1, nonce: 1, expiry: 10e9 }
  );

  const r0 = '0x' + sig0.substring(2, 66);
  const s0 = '0x' + sig0.substring(66, 130);
  const v0 = '0x' + sig0.substring(130, 132);

  await expect(erc721WrapperVotes.wrapBySig(1, 1, 10e9, v0, r0, s0)).to.not.be
    .reverted;

  await expect(erc721WrapperVotes.unwrap(1)).to.not.be.reverted;

  // replay the same signature and nonce does not revert
  await expect(erc721WrapperVotes.wrapBySig(1, 1, 10e9, v0, r0, s0)).to.not.be
    .reverted;
});
```

3.3 Example Exploit

It is common for invested third party entities to create user interfaces on top of existing governance contracts. Such parties will often pay for gas on behalf of end users to perform various governance actions. This helps incentivise governance participation / gathering of delegate votes etc...

If such a voting interface is malicious, they could have users sign gasless transactions with arbitrarily high expiry timestamps. This is something non-technical end users may not notice, and even if they are technical the existence of a nonce in the signature implies one time use. Over time, the malicious entity could "hoard" signatures for use in government manipulation. Here is an example exploit.

Total Voting Supply: 100

1. Over time, attacker collects 90 "unwrap" signatures from participating voters
2. Attacker creates a proposal: "Drain the treasury and send it all to me"
3. Attacker acquires 6 delegate votes
4. Shortly before proposal start time, attacker replays all 90 "unwrap" signatures.

5. The tokens are unwrapped and voting supply is reduced to 10
6. The proposal starts with total supply of 10, attacker has 6 / 10 votes, and votes to pass the proposal
7. Treasury is drained

This is obviously a worst case scenario, and the likelihood of such an exploit occurring is a function of how often users `wrap` and `unwrap` their tokens, and whether they use a third party interface which sets high expiries.

That being said, there are less catastrophic, but still undesirable results from signature replays being available, such as third parties being able to `wrap` tokens which are listed on secondary marketplaces thus removing the listing, and other forms of quorum or NFT price floor manipulation.

3.4 Recommendation

Enforce the nonce is unique in both the `wrapBySig` and `unwrapBySig` functions:

```
require(
    nonce == nonces[signatory]++,
    "ERC721Wrapper::wrapBySig: invalid nonce"
);
```

3.4.1 Additional Recommendation

If the implementer chooses to further reduce the risk of "signature hoarding" outside of replay attacks, the same nonce mapping can be shared for `wrap` `unwrap` and `delegate` so each new action invalidates any previously saved signatures. This encourages interfaces to submit signatures in a timely manner to avoid invalidation.

They may also wish to implement an `invalidatePendingSignatures` function which was callable by end users and increments the nonce for a given address in order to invalidate pending signatures.

4 Unmitigated Timelock Admin

⚠ **Medium Risk**

While conventionally the contract admin will remain the governor contract itself, there are two functions (`_setPendingAdmin`, and `_acceptAdmin`) exposed in Governor.sol which have the ability to update the admin to any address. **This admin has direct access to timelocking functionality and can schedule arbitrary transactions.**

```
/**
  * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to
finalize the transfer.
  * @param newPendingAdmin New pending admin.
  */
function _setPendingAdmin(address newPendingAdmin) external {
    // Check caller = admin
    require(msg.sender == admin, "Governor::_setPendingAdmin: admin only");

    // Save current value, if any, for inclusion in log
    address oldPendingAdmin = pendingAdmin;

    // Store pendingAdmin with value newPendingAdmin
    pendingAdmin = newPendingAdmin;

    // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
    emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);
}

/**
  * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
  */
function _acceptAdmin() external {
    // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
    require(
        msg.sender == pendingAdmin && msg.sender != address(0),
        "Governor::_acceptAdmin: pending admin only"
    );

    // Save current values for inclusion in log
    address oldAdmin = admin;
    address oldPendingAdmin = pendingAdmin;

    // Store admin with value pendingAdmin
    admin = pendingAdmin;

    // Clear the pending value
    pendingAdmin = address(0);

    emit NewAdmin(oldAdmin, admin);
    emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);
}
```

If a malicious actor is able to pass a proposal to grant themselves admin either through honest governance, phishing, or otherwise, they have arbitrary control over the governance contract and can drain any tokens / ether held in the contract.

What is more, there is no throttling on how often a compromised admin might queue transactions via the timelock. This could come in handy if the attacker is ever in a race with honest governance participants to perform some action, potentially making exploits more damaging.

For instance, say the governance treasury has some form of immutable income stream. If a malicious admin has several "withdraw ETH" transactions queued which have already cleared the timelock window, they might have an advantage over honest governance participants to be first to withdraw when new ether lands in the contract. Honest governance participants are throttled to one active proposal at a time per user, and thus have to coordinate among several users in order to queue as many transactions as the malicious admin.

4.1 Recommendation

Consider adding some checks and balances on the contract admin in case it is compromised.

For instance, you might implement a "Pause Guardian" address which has the ability to pause admin execution.

```
bool pausedByGuardian = false;
address pauseGuardian = 0x...;

function _pauseAdminTimelock() {
  require(msg.sender === pauseGuardian, "Only pause guardian can pause");
  pausedByGuardian = true;
}

function _unpauseAdminTimelock() {
  require(msg.sender === pauseGuardian, "Only pause guardian can unpause");
  pausedByGuardian = false;
}

function _executeTransaction(
    address target,
    uint256 value,
    bytes memory data,
    uint256 eta
) public payable returns (bytes memory) {
    require(
        msg.sender == admin,
        "Governor::executeTransaction: Call must come from admin"
    );
    require(!pausedByGuardian, "Timelock paused by guardian");
...
```

Additionally, consider throttling the number of active transactions a timelock admin can queue to one at at time (this may not be necessary if there is a pause guardian).

5 Zero Address Delegation

Medium Risk

5.1 Vulnerability

If a user delegates their votes to the zero address, **their votes will be lost and root tokens locked in the wrapper contract forever.**

This edge case occurs due to auto-delegation in ERC721WrapperVotes.sol contract logic when the delegate is unset. Under certain circumstances described below, this can cause reverts due to underflow. The logic is somewhat nuanced so we also provide a test case below to demonstrate the undesirable behavior:

5.1.1 Test Case

```
it('Disallows unwrap after delegating to the zero address (test passing demos undesirable
behavior)', async function () {
  await testNFT.mint(accounts[0].address, 123);

  // wrap
  await testNFT['safeTransferFrom(address,address,uint256)'](
    accounts[0].address,
    erc721WrapperVotes.address,
    123
  );

  // unwrap works initially
  await expect(erc721WrapperVotes.unwrap(123)).to.not.be.reverted;

  // wrap again
  await testNFT['safeTransferFrom(address,address,uint256)'](
    accounts[0].address,
    erc721WrapperVotes.address,
    123
  );

  const votesBefore = await erc721WrapperVotes.getVotes(accounts[0].address);

  await erc721WrapperVotes.delegate(ethers.constants.AddressZero);

  const votes = await erc721WrapperVotes.getVotes(accounts[0].address);

  expect(votesBefore).to.equal(1);
  expect(votes).to.equal(0);

  // reverts with underflow
  await expect(erc721WrapperVotes.unwrap(123)).to.be.reverted;

  // attempt to delegate to self again (essentially no-op)
  await erc721WrapperVotes.delegate(accounts[0].address);

  // attempt to delegate to another address (reverts)
  await expect(erc721WrapperVotes.delegate(accounts[1].address)).to.be.reverted;

  // still reverts with underflow, token is locked in the contract
  await expect(erc721WrapperVotes.unwrap(123)).to.be.reverted;
});
```

5.1.2 Auto Delegating

If the user has not delegated votes, the contract logic interprets this as delegating to themselves:

```
/**
  * @dev Returns the delegate that `account` has chosen.
  */
function delegates(address account) public view virtual returns (address) {
    if (_delegation[account] == address(0)) {
        return account;
    }
    return _delegation[account];
}
```

5.1.3 Delegate Function

```
/**
  * @dev Delegate all of `account`'s voting units to `delegatee`.
  *
  * Emits events {DelegateChanged} and {DelegateVotesChanged}.
  */
function _delegate(address account, address delegatee) internal virtual {
    address oldDelegate = delegates(account);
    _delegation[account] = delegatee;

    emit DelegateChanged(account, oldDelegate, delegatee);
    _moveDelegateVotes(
        oldDelegate,
        delegatee,
        uint16(userInfos[account].balance)
    );
}

/**
  * @dev Moves delegated votes from one delegate to another.
  */
function _moveDelegateVotes(
    address from,
    address to,
    uint16 amount
) private {
    if (from != to && amount > 0) {
        if (from != address(0)) {
            uint16 oldValue = _delegateCheckpoints[from].push(
                _subtract,
                amount
            );
            emit DelegateVotesChanged(from, oldValue, oldValue - amount);
        }
        if (to != address(0)) {
            uint16 oldValue = _delegateCheckpoints[to].push(_add, amount);
            emit DelegateVotesChanged(to, oldValue, oldValue + amount);
        }
    }
}
```

When a user delegates to the zero address, due to auto-delegation `oldDelegate` will always be interpreted to be a non-zero address, while `delegatee` is still the zero address. As a result, the oldDelegate votes are removed, and no votes are added to the zero address (which is expected).

Once the user has delegated to the zero address, however, their own balance of votes will be zero in `_delegateCheckpoints` state. When they try to delegate or unwrap their tokens, the `oldDelegate` is still interpreted as non zero via `solidity address oldDelegate = delegates(account);` and checkpointing variable underflows when attempting to subtract from `_delegateCheckpoints[oldDelegate]` which has a zero balance.

As a result, unwrapping or re-delegating to another address result in `reverts` and the votes / tokens are essentially lost.

5.2 Exploit Example

This bug can be exploited by attackers who wish to effectively burn tokens or otherwise manipulate governance by censoring certain voters. Burning tokens also reduces the NFT supply which due to scarcity can increase the value of the asset for other token holders. What is more, because this exploit involves interacting directly with the trusted governance contract, misuse and phishing attacks are more feasible.

An example scenario:

1. An important and controversial governance proposal is created with high stakes for participants
2. Malicious actor solicits participants to "delegate to themselves" by clearing their delegation to "ensure their votes are appropriately counted". They may even point to the contract logic which reinforces the idea of "auto-delegation" if your delegation is set to the zero address.
3. Victims tokens are effectively burned and are unable to participate in governance.

A malicious governance UI can accomplish this sort of attack as well via a similar "signature hoarding" strategy as described in the Signature Replay Vulnerability entry.

5.3 Recommendation

Disallow *explicit* delegation to the zero address in the delegation logic. If a user wishes to delegate to themselves they should delegate to their own address. Note that initial auto-delegation will still work with this check in place.

```
require(delegatee != address(0), "Cannot delegate to zero address")
```

5.3.1 Additional Recommendation

The plural `delegates(address)` function implies there can be multiple delegates, consider changing the name to something like `delegateOf(address)`, which more clearly describes the functionality.

II. Low Risk

6 CommunityId Overflow DOS

Low Risk

6.1 Vulnerability

In CommunityToken.sol, the `communityId` and `communityMemberCounter` variables are both `uint24` variables. This low value makes it feasible for a malicious governance NFT holder to deny service from other holders by using up all of the available communityIds.

6.2 Exploit Example

The attacker could accomplish this by transferring their unwrapped nft to a newly generated address, then proceeding to wrap and unwrap with a new address, causing communityMemberCounter to increment on line 152 of CommunityToken.sol. The attacker can repeat this until they have reached the upper bound of community ids available. At that point, no new NFT holders would be able to acquire a CommunityToken due to an overflow revert.

To be clear, on mainnet this would be a *very* expensive and slow attack under current conditions and likely infeasible.

6.2.1 Example economics of the attack:

- Approximate gas cost to wrap, unwrap, transfer tokens: **300,000 gas**.
- To use up the available 2^24 tokenIds would cost: 2^24 * 300,000 = **5,033,164,800,000 gas**. Note: If someone were implementing communityToken w/o disabling transfers, you can increment with only ~50,000 gas via transferFrom so it would be ~1/6 cheaper.

At current gas / eth prices (7PM Saturday, Sep. 17th):

- ETH Price: **$1456.00**
- Gas Price: **3 Gwei**

The total cost to overflow the communityIds would be about **$22M in gas**.

We list this as low risk only if the intent is for the contracts to be used across chains. For instance on Polygon you could overflow communityIds for ~$120,000.

6.3 Recommendation:

Modify `communityId` and `communityMemberCounter` to `uint240` rather than `uint24` . This makes intentional `communityMemberCounter` overflow computationally infeasible, even with several orders of magnitude cheaper gas than any chain currently available. This has the added benefit of more efficient slot packing without the need for padding (which was likely the original intention).

7 Future Proposal Cancellation

Low Risk

7.1 Vulnerability

Any user is able to call `cancel` on a future proposal, causing a `ProposalCanceled` event to be emitted and the `canceled` boolean to be set to `true` for a given proposal before it is even created.

It is important to note that the `canceled` variable will be overridden to `false` when the proposal is actually created.

This works because before a proposal is created, the "proposer" address is the zero address (unset state). The zero address has no delegate votes and thus will be below the proposal threshold, so anyone can cancel the proposal.

Note that this also works for the invalid zero id proposal, although this is less exploitable.

7.2 Test Case

```
it('can cancel a future proposal (undesirable behavior)', async function () {
  await expect(governor.cancel(100))
    .to.emit(governor, 'ProposalCanceled')
    .withArgs(100);
});
```

7.3 Exploit

This bug could be used to confuse government UIs which query cancellation events to display active proposals. While the governance UI implementation is out of the scope of this audit, it is reasonable to believe a UI might hide proposals that had emitted cancellation events. A malicious proposer might use this to pre-cancel their proposal and obfuscate it from governance participants.

As a result, a proposal might receive less participation or review and malicious proposals could be passed that otherwise would have been promptly defeated.

7.4 Recommendation

There are two `state` functions in the governance contract, a high level one that accepts `proposalId` and a lower level helper which accepts `proposal` and `proposalVote` structs.

Throughout the contract, calls are made directly to the lower level function which does not validate `proposalId`, allowing for bugs like cancelling future proposals.

We recommend:

1. Moving `proposalCount >= proposalId` validation into the lower level function
2. Additionally validating `proposalId != 0`

This will cause `cancel` to revert with an invalid proposalId as it calls directly to the lower level state helper.

```
/**
  * @notice Gets the state of a proposal
  * @param proposalId The id of the proposal
  * @return Proposal state
  */
function state(uint256 proposalId) public view returns (ProposalState) {
    Proposal storage proposal = proposals[proposalId];
    ProposalVote storage proposalVote = proposalVotes[proposalId];

    return state(proposal, proposalVote, proposalId);
}

/**
  * @notice Private function that gets the state of a proposal.
  * @param proposal The proposal struct
  * @param proposalVote The proposal vote struct
  * @return Proposal state
  */
function state(Proposal storage proposal, ProposalVote storage proposalVote, uint256 proposalId)
    private
    view
    returns (ProposalState)
{
    require(
        proposalCount >= proposalId,
        "Governor::state: invalid proposal id"
    );
    if (proposal.canceled) {
        return ProposalState.Canceled;
    } else if (block.timestamp <= proposal.startTime) {
        return ProposalState.Pending;
    } else if (block.timestamp <= proposal.endTime) {
        return ProposalState.Active;
    } else if (
        proposalVote.forVotes <= proposalVote.againstVotes ||
        (proposalVote.forVotes +
            proposalVote.againstVotes +
            proposalVote.abstainVotes) <
        proposal.quorum
    ) {
        return ProposalState.Defeated;
    } else if (proposal.executed) {
        return ProposalState.Executed;
    } else if (
        block.timestamp >=
        proposal.endTime + TIMELOCK_DELAY + TIMELOCK_GRACE_PERIOD
    ) {
        return ProposalState.Expired;
    } else if (block.timestamp < proposal.endTime + TIMELOCK_DELAY) {
        return ProposalState.Queued;
    } else {
        return ProposalState.Executable;
    }
}
```

III. Informational

8 Admin Events Convention

Informational

Events are a great way for back end systems and user interfaces to stay up to date with the latest governance changes. They also serve as audit logs for the contract.

In Governor.sol, when a change is made to a governance parameter, an event is emitted, e.g. `VotingPeriodSet` and `ProposalThresholdSet`.

PVFDGovernance, however, breaks this convention and does not emit events when governance parameters around quorum are modified.

8.1 Recommendation

Declare and emit `BaseSupplySet(uint16 count)` and `QuorumBipsSet(uint16 bips)` events when modifying governance quorum parameters to inform any systems listening for events.

9 Admin Modifier

Informational

Both Governor.sol and PVFDGovernor.sol frequently gate admin functions with the same block of repeated code, always at the beginning of the function:

```
require(
    msg.sender == admin,
    "Contract::function: admin only"
);
```

This is a good candidate for a solidity modifier to improve cleanliness and readability of code.

9.1 Recommendation

Instead of repeating the same block, consider implementing a modifier to use across functions.

```
modifier onlyAdmin() {
    require(
        msg.sender == admin,
        "Admin only"
    );
    _;
}
```

10 Cast Votes Readability

Informational

`castVoteInternal` is one of the most important governance functions in the codebase. It is worth ensuring the code is not only functional, secure, and bug-free, but also highly readable, even self documenting.

Currently, the function spans over 70 lines of logic and has mirrored conditionals on lines 481 and 524:

```
481 if (voterCommunityId < 209) {

...

524 if (voterCommunityId < 209) {
```

10.1 Recommendation

The readability of this critical governance function might be improved by breaking the function logic into smaller discrete components. Specifically, the code might benefit from implementing helpers for packing hasVoted in `lowerVotes` vs. `upperVotes`.

See the below alternative with smaller, discrete functions

```solidity
function setHasVotedLower(
    ProposalVote storage proposalVote,
    uint256 voterCommunityId
) internal {
    // bring lowerVotes into the stack
    uint208 lowerVotesWord = proposalVote.lowerVotes;
    // Bit flag at index voterCommunityId - 1 (since id starts at 1)
    // indicates if the voter has voted on this proposal.
    bool voted = lowerVotesWord & (1 << (voterCommunityId - 1)) ==
        (1 << (voterCommunityId - 1));

    require(!voted, "Governor::castVoteInternal: voter already voted");

    // Using lower votes word.
    proposalVote.lowerVotes =
        lowerVotesWord |
        uint208(1 << (voterCommunityId - 1));
}

function setHasVotedUpper(
    ProposalVote storage proposalVote,
    uint256 voterCommunityId
) internal {
    // Community id is too large to fit in lowerVotesWord. Therefore,
    // pull from the upperWords. Upper words indecies start at 209.
    uint256 wordIndex = (voterCommunityId - 209) / 256;
    // Get bit index within the word.
    uint256 bitIndex = (voterCommunityId - 209) % 256;
    uint256 upperVotesWord = proposalVote.upperVotes[wordIndex];

    // Bit flag at index bitIndex indicates if the voter has voted on this proposal.
    bool voted = upperVotesWord & (1 << bitIndex) == (1 << bitIndex);

    require(!voted, "Governor::castVoteInternal: voter already voted");

    // Using up votes word.
    proposalVote.upperVotes[wordIndex] = upperVotesWord | (1 << bitIndex);
}

function castVoteInternal(
    address voter,
    uint256 proposalId,
    VoteType support
) internal returns (uint16) {
    ProposalVote storage proposalVote = proposalVotes[proposalId];
    uint256 voterCommunityId = token.getCommunityId(voter);

    require(
        voterCommunityId != 0,
        "Governor::castVoteInternal: voter invalid"
    );

    if (voterCommunityId < 209) {
        setHasVotedLower(proposalVote, voterCommunityId);
    } else {
        setHasVotedUpper(proposalVote, voterCommunityId);
    }

    Proposal storage proposal = proposals[proposalId];
    require(
        state(proposal, proposalVote) == ProposalState.Active,
        "Governor::castVoteInternal: voting is closed"
    );
    uint16 votes = token.getPriorVotes(voter, uint40(proposal.startTime));

    if (support == VoteType.Against) {
        proposalVote.againstVotes += votes;
    } else if (support == VoteType.For) {
        proposalVote.forVotes += votes;
    } else {
        // must be abstain
        proposalVote.abstainVotes += votes;
    }

    return votes;
}
```

Note: The above alternative does add some gas overhead due to additional jump ops, but ends up being cheaper in the happy case of the communityId being `< 209` because it doesn't declare unnecessary variables such as `wordIndex`.

A similar approach might be used for `hasVoted` function which has similar logic.

11 Checkpoints Documentation

Informational

The checkpoints documentation copied over from Open Zeppelin is outdated and still refers to block numbers throughout the library.

```
/**
  * @dev Returns the value at a given block number. If a checkpoint is not available at that
block, the closest one
  * before it is returned, or zero otherwise.
  */
function getAtTime(History storage self, uint40 timestamp)
```

11.1 Recommendation

Update the documentation to reflect the implementation which uses block timestamps.

12 Checkpoints Optimizations

Informational

We noticed three small gas inefficiencies in the checkpoints library.

1. The `getAtTime` function calculates an average with `(low & high) + (low ^ high) / 2` which could be cheaper with `(low & high) + ((low ^ high) >> 1)`. Right shift is cheaper than division and achieves the same result of divide by two.
2. In the `push` function `pos > 0` can be modified to `pos != 0` as pos is unsigned and inequality is cheaper than comparison.
3. Normal binary search may not be the most efficient algorithm for finding the desired past checkpoint in practice. For the governance checkpointing use case, old checkpoints are likely less relevant as governance proposals have finite time periods before they are either executed or defeated. Thus you may save users gas by choosing an algorithm that is more biased toward recent blocks. Open zeppelin exposes such an algorithm in their latest checkpointing lib via `getAtProbablyRecentBlock` which uses `len - Math.sqrt(len);` for the midpoint on large checkpointing sets. Note that making such a change may not be worth it if you don't foresee a high frequency of delegation as it adds some overhead.

13 Delegate Centralization

Informational

NFTs typically have smaller supplies than ERC20 governance tokens. What is more, if root token NFTs have other non governance utilities, end users may not immediately opt into governance.

As such, there could be a higher risk for delegate centralization than most governance use cases. Even with benevolent delegates, centralization poses risk in the case that an important wallet with many delegate votes is compromised. Having votes spread amongst more users allows for better checks and balances and an overall more secure system.

To combat delegate centralization, the implementer could consider introducing limits on the number of delegate votes a given address can receive. Something to the tune of the following pseudo-code:

```
maxDelegates = ....

delegate(toAddress, toAmount) {
  require(currentDelegates + toAmount < maxDelegates, "Max delegates already reached");
}
```

Note: Such an addition adds scope and bug surface area to the contract, which may be undesirable.

14 Duplicate EIP712 Logic

Informational

ERC721Wrapper.sol inherits Open Zeppelin's `EIP712`, then proceeds to re-implement much of the storage and logic that the parent contract provides.

OZ's abstract contract `EIP712` handles storing chainId and the domain typehash. It also has helpers for calculating domain separators (even in the event of a blockchain fork), as well as creating the signature digest for verification.

14.1 Recommendation

1. Remove the variables `_CHAIN_ID`, `_DOMAIN_SEPARATOR`, `_DOMAIN_TYPEHASH`, as well as the contents within `_domainSeparator()` from `ERC721Wrapper` contract
2. Call on `EIP712._domainSeparatorV4()` instead of the implemented domain separator.
3. Optionally utilize `_hashTypedDataV4(bytes32 structHash)` to calculate the typed hash for recovery in `unwrapBySig` `wrapBySig` and `delegateBySig` in the inheriting contract

Or alternatively, do not inherit OZ's `EIP712`

15 ERC721Receiver Omission

Informational

There is an argument to be made for omitting ERC721Receiver from ERC721Wrapper. The idea is for users to be able to send their tokens to the contract address to initiate wrapping their token for governance participation. One potential risk is that this could lead to confusion with end users.

Specifically non-technical users may think they can simply send their NFT to the contract address and may not understand the nuance between `safeTransfer` and `transfer` (wrapping and accounting logic can only be implemented on `safeTransfer`).

If any user transfers their NFT to the communityToken contract via `transferFrom`, that NFT will be locked in the contract and wrapping will not be accounted for. Only having one explicit method for wrap / unwrap might help prevent confusion on the matter.

Note This risk is highly theoretical. With good communication to end users this could be a non-issue, but is worth mentioning.

16 Future Transaction Cancel

Informational

An admin address can cancel a transaction before it exists using the timelock functionality in Governor.sol.

```
function _cancelTransaction(
    address target,
    uint256 value,
    bytes memory data,
    uint256 eta
) public {
    require(
        msg.sender == admin,
        "Governor::cancelTransaction: Call must come from admin"
    );

    bytes32 txHash = keccak256(abi.encode(target, value, data, eta));
    queuedTransactions[txHash] = false;

    emit CancelTransaction(txHash, target, value, data, eta);
}
```

By calling `_cancelTransaction` on a nonexistent transaction hash, a `CancelTransaction` event is emitted which might confuse governance UIs.

16.1 Recommendation

Validate the transaction is actually queued before enabling cancel.

`require(queuedTransactions[txHash], "Transaction not Queued")`

17 Get Actions Validation

Informational

In Governor.sol the `getActions` function does not validate that the proposal id is valid, and will typically just return zero bytes for `targets`, `values` and `calldatas`.

```
function getActions(uint256 proposalId)
    external
    view
    returns (
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas
    )
{
    Proposal storage p = proposals[proposalId];
    return (p.targets, p.values, p.calldatas);
}
```

17.1 Recommendation

Revert if the `proposalId` is invalid when `getActions` is called.

```
require(proposalCount >= proposalId && proposalId != 0, "Invalid proposal Id");
```

18 Governance UI

Informational

The current implementation of PVFDGovernance is incompatible with third party governance user interfaces such as Tally.

Supporting different third party interfaces creates redundancy for the UI layer and reduces centralization risk that could damage governance. It also helps mitigate the damage which can be caused by certain off chain attacks such as XSS and domain hijacking.

18.1 Recommendation

Consider modifying the contract structure to be compatible with third party UIs and follow established Compound Governance Bravo fork conventions.

Some of the established conventions can be found here:

https://docs.tally.xyz/user-guides/tally-contract-compatibility/compound-bravo-style

Also see token conventions:

https://docs.tally.xyz/user-guides/tally-contract-compatibility/tokens-erc20-and-nfts

19 Missing Delegate Validation

Informational

In Governor.sol `castVoteInternal` validates that the voter must have a `communityId`

```
require(
    voterCommunityId != 0,
    "Governor::castVoteInternal: voter invalid"
);
```

But in ERC721WrapperVotes.sol the end user is able to delegate votes to any address, even if they do not have a communityId:

```
function _delegate(address account, address delegatee) internal virtual {
    address oldDelegate = delegates(account);
    _delegation[account] = delegatee;

    emit DelegateChanged(account, oldDelegate, delegatee);
    _moveDelegateVotes(
        oldDelegate,
        delegatee,
        uint16(userInfos[account].balance)
    );
}
```

19.1 Recommendation

When a user delegates their votes, validate that the delegatee is a valid community member.

```
  require(getCommunityId(delegatee) != 0, "Delegatee cannot vote");
```

Note, this validation also would also alleviate issues around zero address delegation.

20 Proposal Signature Omission

Informational

When a proposal is created, the `calldatas`, `targets`, and `values` are stored on chain and emitted in an event, but the `signatures` of the functions to be called are neither stored nor emitted.

```
proposals[proposalId] = Proposal({
    startTime: uint40(startTime), // time will be less than 2**40 until year 36812
    endTime: uint40(endTime),
    canceled: false,
    executed: false,
    quorum: currentQuorum(),
    proposer: msg.sender,
    targets: targets,
    values: values,
    calldatas: calldatas
});

latestProposalIds[msg.sender] = proposalId;

emit ProposalCreated(
    proposalId,
    msg.sender,
    targets,
    values,
    calldatas,
    startTime,
    endTime,
    description
);
```

Function signatures are useful to know what is going on under the hood in a proposal. By omitting them from the proposal events and on chain object, although gas is saved, there may be some unintended consequences downstream.

Namely:

1. It breaks compatibility with governance UIs such as Tally which display function selectors, see their docs on event signatures here
2. Even if governance UI compatibility is is not a concern, omitting selectors makes it easier for proposers to be dishonest about what is happening in their proposal. E.G. proposing with an innocuous proposal description when the underlying function calls are malicious.

20.0.1 Additional Information

Signatures, while helpful for transparency are not foolproof. The end selector used to make a function call is only 4 bytes, which, being a relatively small space can have collisions between two different signatures that hash to the same end selector.

Thus attackers who precompute colliding signatures could still mislead proposer reviewers by including a colliding signature that is different from the real function that is actually implemented in the contract.

Additionally, proposers can still choose to omit the function selector in favor of encoding their own calldata (or the target contract's fallback function).

All of this is to say, there is not a substitute to inspecting the transaction target itself in the context of the proposed transaction.

20.1 Recommendation

- Keep signatures in the Proposal Event for transparency, it may not be necessary to store them on chain.
- Make the `propose` function which does not accept signatures internal, as to force proposers to go through the function which includes signatures and encourage providing function signatures.
- More generally, follow guidelines around function / event signatures by existing governance UIs if you wish to maintain compatibility with them, e.g. https://docs.tally.xyz/user-guides/tally-contract-compatibility/compound-bravo-style#event-signatures

21 Quorum Documentation

Informational

Compound governance quorum is determined by the following condition:

`proposal.forVotes < quorumVotes`

Where `quorumVotes` is an immutable constant.

By contrast, PVFD fork quorum is determined by

```
(proposalVote.forVotes + proposalVote.againstVotes + proposalVote.abstainVotes) < proposal.quorum
```

Where `proposal.quorum` is determined at proposal time as a function of token supply and quorum bips set by meta-governance.

The break from Compound convention of quorum being solely determined by `forVotes` is worth documenting at the `state` function level as well as in communications to users. End users might have expectations around what determines quorum that differ from this implementation.

It is also worth noting that governance UIs such as Tally expect a `quorumVotes` constant for their quorum display.

21.1 Recommendation

- Document the unique quorum calculation in the `state` function in natspec format
- If perfect compatibility with governance UIs is desired, there may need to be more substantial revision to the quorum mechanic.

22 Signature Malleability

Informational

Signature malleability occurs when there are multiple valid signatures for the same data and public key.

`ecrecover` allows for malleable signatures. `(r,s,v)` and `(r,s',v)` from two "Y" points for a given "X" on the elliptic curve.

`ecrecover` is used throughout the codebase. No meaningful exploits were found as a result of signature malleability.

22.1 Recommendation

Even so, consider using Open Zeppelin's ECDSA library to recover signature public keys, as they have an explicit check which prevents malleability.

23 Timelock Event Indexing

Informational

The `ExecutedTransaction` event has `txHash` indexed

```
event ExecuteTransaction(
    bytes32 indexed txHash,
    address indexed target,
    uint256 value,
    bytes data,
    uint256 eta
);
```

Indexed event parameters cost more gas, and as such should only be used if useful for search filtering.

The majority of the time `ExecuteTransaction` is emitted with a hard coded zero txHash.

```
emit ExecuteTransaction(0, target, value, callData, proposalEta);
```

Which is not particularly useful for search.

23.1 Recommendation

Remove `indexed` from `txHash` or remove `txHash` altogether. Alternatively, implement two different `ExecuteTransaction` events, one with indexing for timelock functionality and one without for proposals.

24 Token Receiver Documentation

Informational

The documentation for `onERC1155BatchReceived` and `ERC1155Received` are swapped

```
/// @notice Receive ERC1155
function onERC1155BatchReceived(
    address,
    address,
    uint256[] calldata,
    uint256[] calldata,
    bytes calldata
) external pure virtual returns (bytes4) {
    return IERC1155Receiver.onERC1155BatchReceived.selector;
}

/// @notice Receive ERC1155 batch
function onERC1155Received(
    address,
    address,
    uint256,
    uint256,
    bytes calldata
) external pure virtual returns (bytes4) {
    return IERC1155Receiver.onERC1155Received.selector;
}
```

24.1 Recommendation

Appropriately modify the documentation:

```
/// @notice Receive ERC1155 batch
function onERC1155BatchReceived(
    address,
    address,
    uint256[] calldata,
    uint256[] calldata,
    bytes calldata
) external pure virtual returns (bytes4) {
    return IERC1155Receiver.onERC1155BatchReceived.selector;
}

/// @notice Receive ERC1155
function onERC1155Received(
    address,
    address,
    uint256,
    uint256,
    bytes calldata
) external pure virtual returns (bytes4) {
    return IERC1155Receiver.onERC1155Received.selector;
}
```

25 Uint8 Comparison

Informational

In ERC721Wrapper.sol `wrap` and `unwrap` functions, the `for` loop counter is a `uint8`

```
function wrapMany(uint256[] calldata tokenIds) external {
    for (uint8 i; i < tokenIds.length; i++) {
        _wrap(msg.sender, tokenIds[i]);
    }
    _onTokenWrap(msg.sender, uint16(tokenIds.length));
}
```

`uint8` is more expensive than `uint256` because it is implicitly upcast to `uint256` for each comparison to `tokenIds.length`.

25.1 Recommendation

- Change `uint8` counters to `uint256`
- If the intent is to cap the number of wrappable NFTs to 255 via overflow reverts, this intent should be documented accordingly with a comment.

26 Unused Queue Event

Informational

The `event ProposalQueued(uint256 id, uint256 eta);` is declared in GovernorEvents.sol but never emitted.

26.1 Recommendation

Remove the unused event.

27 Unused Quorum Variable

Informational

In Governor.sol there is an unused constant `QUORUM_VOTES`.

27.1 Recommendation

Remove the unused constant.

28 Variable Shadowing

Informational

Variable shadowing occurs when a local variable has the same name as a state variable. This is discouraged as it reduces the readability of the code and is prone to errors. Best practice is to avoid shadowing not only in a given contract, but also in its inheriting contracts.

The `name` and `symbol` constructor arguments shadow inherited state variables from `CommunityToken` in the following contracts:

- `ERC721Wrapper`
- `ERC721WrapperVotes`
- `PVFDWrapperVotes`

This was not found to cause any bugs, but does inhibit the readability of the contract.

28.1 Recommendation

Rename the constructor arguments and their usages to `name_` and `symbol_` in the above contracts to eliminate variable shadowing.

29 Voting Delay Boundary

Informational

The lower boundary for voting delay is 1 second:

```
/// @notice The min setable voting delay
uint40 public constant MIN_VOTING_DELAY = 1 seconds;
```

The idea behind the voting delay is to enable users time to react to a new proposal and allocate their votes accordingly before voting balances are tallied at proposal start time.

One second does not offer enough time for human reaction. Having voting delay at such a low value could introduce unwanted game theory around proposal creation timing. Proposers would likely attempt to game the voting distribution by proposing when they have an advantage.

29.1 Recommendation

Increase the `MIN_VOTING_DELAY` lower boundary to something that affords human reaction, like 1 hour (or even 12 hours to cover all timezones).

In practice, `votingDelay` is determined via meta-governance, but this is still a useful boundary to maintain fair global access to governance.

30 Zero Address Transfer

Informational

In CommunityToken.sol `transferFrom` and `safeTransferFrom` do not validate against transferring `to` the zero address. This in theory would allow for end users to burn tokens without going through an explicit `burn` function. It would also cause the zero address to gain a non-zero token balance. Validating against the zero address is also described in the ERC721 specification.

In practice transfers are disabled by the inheriting `ERC721WrapperVotes.sol` so this is a non issue for this use case of `CommunityToken`.

30.1 Recommendation

Even so, we recommend validating against the zero address in both `transferFrom` and `safeTransferFrom`:

```
require(to != address(0), "Cannot transfer to zero addresss");
```

31 Disclaimer