Prepared for
Weso
Beefy

Prepared by
Malte Leip
Vlad Toie
Zellic

February 28, 2024

Beefy UniswapV3

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Beefy from February 26th to February 28th, 2024. During this engagement, Zellic reviewed Beefy UniswapV3's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- How does Beefy perform deposits and withdrawals? What about the issuance of shares?
- Is the transition across ticks done safely? What are the conditions for determining if a transition is safe?
- How is TWAP used in the system? What are the conditions for determining if a TWAP is safe?
- What tokens are supported by Beefy? How are they handled? Are their decimals handled correctly?

1.3. Non-goals and Limitations

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

- Usage of the vault contract with a different strategy contract and vice versa
- Front-end components
- Infrastructure relating to the project
- Key custody

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

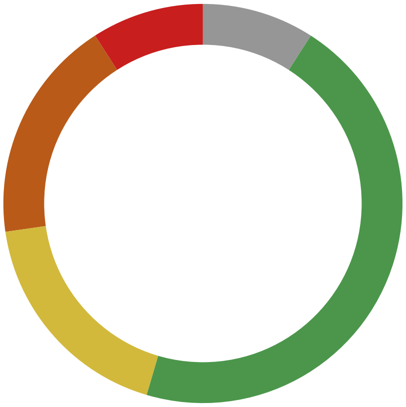
1.4. Results

During our assessment on the scoped Beefy UniswapV3 contracts, we discovered 11 findings. One critical issue was found. Two were of high impact, two were of medium impact, five were of low impact, and the remaining finding was informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	2
🟨 Medium	2
🟩 Low	5
◼ Informational	1



2. Introduction

2.1. About Beefy UniswapV3

Beefy is a decentralized, multichain yield optimizer that allows its users to earn compound interest on their crypto holdings. Beefy earns you the highest APYs with safety and efficiency in mind.

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 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 codebase 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Beefy UniswapV3 Contracts

Repository	https://github.com/beefyfinance/experiments ↗
Version	experiments: `9b54aee1d8098966f582d67ff054e011897c5f30`
Programs	• StrategyFactory • StrategyPassiveManagerUniswap • BeefyVaultConcLiq • BeefyVaultConcLiqFactory
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 0.8 person-weeks. The assessment was conducted over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Malte Leip
Engineer
malte@zellic.io ↗

Vlad Toie
Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 26, 2024	Kick-off call
February 26, 2024	Start of primary review period
February 28, 2024	End of primary review period

3. Detailed Findings

3.1. Price deviation for calm period detection is done by exponentiating, not multiplying

Target	StrategyPassiveManagerUniswap		
Category	Business Logic	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

A general issue with using the current Uniswap price as a price oracle is that these prices can be manipulated by an attacker during their transaction, possibly using flash loans.

To protect against such price manipulation, deposits are only allowed during a calm period in which prices have not recently moved by much. This is implemented by the `onlyCalmPeriods` modifier, which should revert when the relative change of the current price compared to the time-weighted average price over the last 60 seconds is too large.

This is the relevant code:

```
/// @notice Modifier to only allow deposit/harvest actions when current price
    is within 1% of twap.
modifier onlyCalmPeriods() {
    int24 tick = currentTick();
    int56 twapTick = twap();

    int56 upperBound = deviation + deviationThreshold;
    int56 lowerBound = deviationThreshold - deviation;

    // Calculate if greater than deviation % from twap and revert if it is.
    if (tick >= 0) {
        if (tick > twapTick * upperBound / deviationThreshold || tick <
    twapTick * lowerBound / deviationThreshold) revert NotCalm();
    } else if (tick < twapTick * upperBound / deviationThreshold || tick >
    twapTick * lowerBound / deviationThreshold) revert NotCalm();
    _;
}
```

This code actually operates at the level of *ticks* rather than *price*. In Uniswap V3, the tick associated to a price is defined by $tick = \log_{1.0001}(price)$. In `onlyCalmPeriods`, the variable `tick` then holds the tick corresponding to the current tick, while `twapTick` holds the time-weighted arithmetic mean tick over the last 60 seconds. This tick corresponds to the time-weighted geometric mean price over the last 60 seconds, which we will call `twapPrice`. For more details on this, see the

Uniswap V3 white paper ↗, section 5.2.

The check in `onlyCalmPeriods` has two variants to deal with signs. To simplify exposition, let us assume for the moment that `tick` and `twapTick` are both positive. Then what is checked (i.e., it reverts if this does not hold) is the following, where we use $d = deviation/deviationThreshold$ (which is 0.01 by default).

$$\begin{aligned}&\frac{deviationThreshold - deviation}{deviationThreshold} \cdot twapTick \leq tick \\ &\qquad\qquad\qquad\leq \frac{deviationThreshold + deviation}{deviationThreshold} \cdot twapTick \\ \iff &(1-d) \cdot twapTick \leq tick \leq (1+d) \cdot twapTick \\ \iff &1.0001^{(1-d)\cdot twapTick} \leq 1.0001^{tick} \leq 1.0001^{(1+d)\cdot twapTick} \\ \iff &\left(1.0001^{twapTick}\right)^{1-d} \leq price \leq \left(1.0001^{twapTick}\right)^{1+d} \\ \iff &twapPrice^{1-d} \leq price \leq twapPrice^{1+d}\end{aligned}$$

In order to check for the relative change of $price$ compared to $twapPrice$ of at most, for example, 1%, using $d = 0.01$ for this, the check should instead have been this:

$$(1-d) \cdot twapPrice \leq price \leq (1+d) \cdot twapPrice$$

A consequence of exponentiating with $1 - d$ and $1 + d$ instead of multiplying with it is that the relative change allowed changes with $twapPrice$. Here is a table with some example values where $d = 0.01$, the default value, with percentage changes in parentheses:

$twapPrice$	minimum $price$ allowed	maximum $price$ allowed
1	1.0 (0.00%)	1.0 (0.00%)
2	1.9862 (-0.69%)	2.0139 (0.70%)
10	9.77 (-2.28%)	10.23 (2.33%)
0.1	0.10233 (2.33%)	0.09772 (-2.28%)
$1,000,000,000,000$	$758,577,575,029$ (-24.14%)	$1,318,256,738,556$ (31.83%)

The above table was generated with this Python code:

```
for twap in (1, 2, 10, 0.1, 10**12):
    min_p = twap**0.99
    max_p = twap**1.01
    min_perc = ((min_p / twap) - 1) * 100
    max_perc = ((max_p / twap) - 1) * 100
    print(f'| `L!${twap}$` | `L!${min_p:,}$` (`L!${min_perc:.2f}$`%) |
    `L!${max_p:,}$` (`L!${max_perc:.2f}$`%) |')
```

Note that the relative change allowed in the positive direction can be calculated as follows when $twapPrice$ is above 1:

$$\frac{twapPrice^{1+d}}{twapPrice} - 1 = twapPrice^{d} - 1 = \left(twapPrice^{-1}\right)^{-d} - 1 = \frac{\left(twapPrice^{-1}\right)^{1-d}}{twapPrice^{-1}} - 1$$

As the latter value is the relative change allowed in the negative direction when the time-weighted average price is $twapPrice^{-1}$, this means the situation is symmetrical (in the multiplicative sense) around 1.

The relative change allowed being zero when $twapPrice = 1$, and being bigger the further away $twapPrice$ is from 1 has two different negative consequences.

Firstly, when $twapPrice = 1$ or is very close to it, the `onlyCalmPeriods` modifier will revert on even very tiny deviations from that price. A time-weighted average price of very close to 1 is not unrealistic in pools for two different USD stablecoins, for example. The StrategyPassiveManagerUniswap contract may then disallow deposits a lot more often than desired.

Secondly, when $twapPrice = 1$ is very large or very small, then the `onlyCalmPeriods` check will be much weaker than expected, allowing the possibility of an attacker to profit from price manipulations after all. A very large or very small price is not unrealistic, for example the DAI stablecoin has 18 decimals while USDC has 6 decimals, so the price of the DAI/USDC pool will tend to be around 10^{-12}; in which case, with $d = 0.01$, a relative deviation of up to 31.83% is allowed in one direction, as can be read off from the above table.

Note how in the current implementation, the behavior of the `onlyCalmPeriods` modifier depends significantly on the amount of decimals the tokens have. As decimals should be an implementation detail and not play a role in financial logic, this is an indication that the current checks are incorrect.

Impact

If the time-weighted average price is close to 1, deposits will be disallowed more often than intended.

If the time-weighted average price is far away[1] from 1 on a logarithmic scale, an attacker manipulating the Uniswap pool's price might be able to drain the pool of part of its value within one transaction while making a profit. See 4.1. ↗ for a more detailed explanation of this, where we estimate that in the context of a DAI/USDC Uniswap pool with a fee of 0.01%, it is profitable to carry out an attack that drains value from the vault/strategy contract when the strategy's liquidity is at least about 0.001 of the other liquidity providers' liquidity in the manipulated range and as long as the relative deviation exceeds about 25.02%. If the liquidity ratio is 0.01 instead of 0.001, then 2.06% would suffice. Given that in this case we arrived at the relative deviation allowed being up to 31.83%, this leaves ample room for minor inaccuracies in the estimation as well as fees for flash loans.

[1] In a logarithmic sense (i.e., 10 is as far away from 1 as 1/10 is).

Recommendations

Consider changing `onlyCalmPeriods` to check for relative change in the price rather than the ticks. Equivalently, check for absolute change in the ticks. This is equivalent:

$$
\begin{aligned}
& (1-d)\cdot twapPrice \le price \le (1+d)\cdot twapPrice \\
\iff\ & \log_{1.0001}((1-d)\cdot twapPrice) \le \log_{1.0001}(price) \le \log_{1.0001}((1+d)\cdot twapPrice) \\
\iff\ & \log_{1.0001}(twapPrice) + \log_{1.0001}(1-d) \le \log_{1.0001}(price) \\
& \qquad \le \log_{1.0001}(twapPrice) + \log_{1.0001}(1+d) \\
\iff\ & twapTick + \log_{1.0001}(1-d) \le tick \le twapTick + \log_{1.0001}(1+d)
\end{aligned}
$$

Note that $\log_{1.0001}(1-d)$ and $\log_{1.0001}(1+d)$ are constants (they do not depend on `twapPrice` or `tick`), so there is no need to calculate logarithms on chain for this check.

Remediation

This issue has been acknowledged by Beefy, and fixes were implemented in the following commits:

- `a7c127a2` ↗
- `e6b02aaf` ↗
- `cb5fd465` ↗

3.2. TWAP interval too short

Target	StrategyPassiveManagerUniswap		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The TWAP represents the time-weighted average price over a certain time interval. This time interval average is used when determining whether a period is calm or not. The current implementation uses a 60-second interval, which represents approximately four, almost five, blocks on the Ethereum mainnet, where the block time is 12 seconds on average.

This interval is too short, as an attacker could theoretically manipulate the price within this time frame. A longer interval would make it harder for an attacker to manipulate the price and would make the system more robust against flash-loan attacks.

Additionally, the current implementation hardcodes the interval, which makes it difficult to change in the future. Depending on the token and the market, the interval should be adjusted to a value that makes it hard for an attacker to manipulate the price within that time frame.

Impact

As the time-frame necessary for an attacker to manipulate the price is too short, the system is more prone to manipulation attacks. This could lead to a loss of funds for the users of the system.

Recommendations

We recommend increasing the base-time interval to a value that makes it hard for an attacker to manipulate the price within that time frame. Additionally, we recommend making the interval adjustable, so that it can be changed in the future if necessary.

```
function updateTwapInterval(uint32 _interval) external onlyOwner {
    twapInterval = _interval;
}

function twap() public view returns (int56 twapTick) {
    uint32[] memory secondsAgo = new uint32[](2);
    secondsAgo[0] = 60;
    secondsAgo[0] = twapInterval;
    secondsAgo[1] = 0;

    (int56[] memory tickCuml,) = IUniswapV3Pool(pool).observe(secondsAgo);
    twapTick = (tickCuml[1] - tickCuml[0]) / 60;
    twapTick = (tickCuml[1] - tickCuml[0]) / twapInterval;
}
```

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit bbde6268 ↗.

3.3. Balances exhibit discontinuous behavior

Target	StrategyPassiveManagerUniswap		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The strategy contract's `balances()` function exhibits discontinuous behavior (that is, the function value "jumps") stemming from two different sources.

The first is in the `balances()` function itself:

```
function balances() public view returns (uint256 token0Bal, uint256 token1Bal)
    {
    (uint256 thisBal0, uint256 thisBal1) = balancesOfThis();
    (uint256 poolBal0, uint256 poolBal1,,,,) = balancesOfPool();
    (uint256 locked0, uint256 locked1) = lockedProfit();

    uint256 total0 = thisBal0 + poolBal0 - locked0;
    uint256 total1 = thisBal1 + poolBal1 - locked1;
    uint256 unharvestedFees0 = fees0;
    uint256 unharvestedFees1 = fees1;

    // If pair is so imbalanced that we no longer have any enough tokens to pay
    fees, we set them to 0.
    if (unharvestedFees0 > total0) unharvestedFees0 = 0;
    if (unharvestedFees1 > total1) unharvestedFees1 = 0;

    // For token0 and token1 we return balance of this contract + balance of
    positions - locked profit - feesUnharvested.
    return (total0 - unharvestedFees0, total1 - unharvestedFees1);
}
```

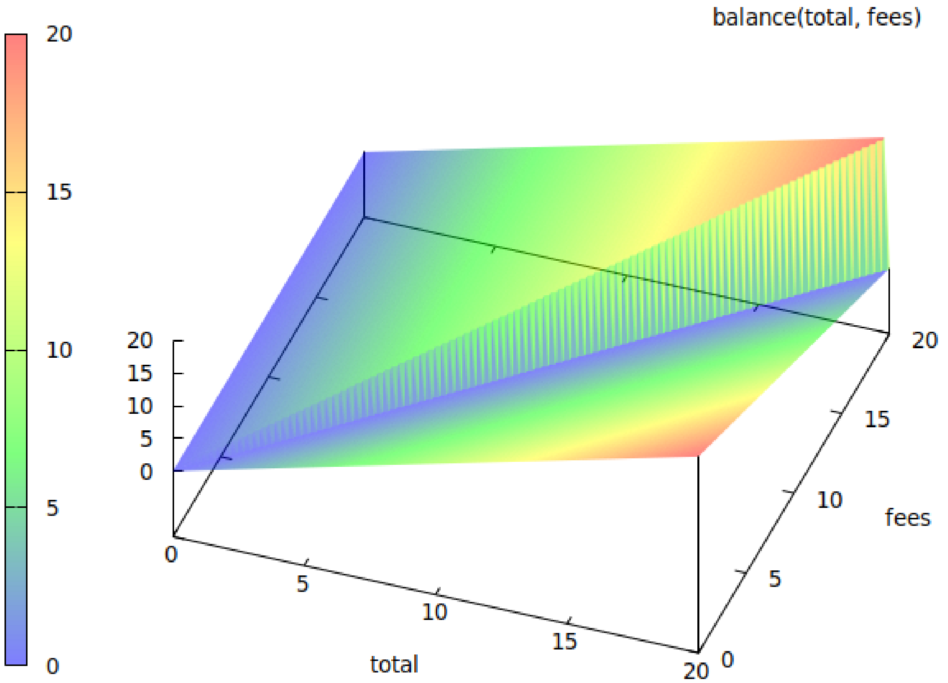
The crucial lines here are the two conditional adjustments to `unharvestedFees0` and `unharvestedFees1`.

```
if (unharvestedFees0 > total0) unharvestedFees0 = 0;
if (unharvestedFees1 > total1) unharvestedFees1 = 0;
```

We will drop the 0 and 1 suffix here, as the two cases are completely analogous. Then the balance returned is of the form

$$balance = \begin{cases} total - fees & \text{if } total \geq fees \\ total & \text{if } total < fees \end{cases}$$

If, say, $fees$ stayed fixed and a large number, and $total$ approaches $fees$ from below, then the value of $balance$ will also approach $fees$ and then suddenly drop to 0 when $total = fees$. The following graph of this function depicts this discontinuity as well.



Such discontinuities are dangerous as an attacker might engineer a situation that is very close to the discontinuity and is then able to make the balance jump by only using a comparatively small amount of funds, for example a single Wei.

In this particular case, an attacker can also indeed engineer such a situation. If we start out in the case where `fees0 < total0`, as will usually be the case, then the attacker can buy token0 for token1 in the Uniswap pool. Doing this will not change `fees0`, and will decrease `total0`. Eventually, the attacker would have converted the strategy's entire position to token1, so that `total0 = 0`, but we would still have `fees0 > 0`. As `total0` acts sufficiently continuous,[2] there is thus a point in between at which the attacker can stop, not having converted quite all of the strategy's token0 to token1, at which `total0` is just under `fees0` (e.g., just one Wei smaller).[3]

By stopping at such a spot, the attacker is now in a situation in which, by trading a miniscule amount of tokens on the Uniswap pool, they can repeatedly move back and forth between the discontinuity. Other ways to influence `fees0` or `total0` can also cause a jump across the discontinuity, for example by sending the strategy contract a single Wei of token0.

The most dangerous entry point to profit from such an attack would be in the vault's `deposit` function. To calculate how much the caller contributed, and thus how many shares they should get, that function compares the difference in `balances()` from before to after the caller's deposit:

```
(uint256 _bal0, uint256 _bal1) = balances();
if (_amount0 > 0) IERC20Upgradeable(token0).safeTransferFrom(msg.sender,
    address(strategy), _amount0);
if (_amount1 > 0) IERC20Upgradeable(token1).safeTransferFrom(msg.sender,
    address(strategy), _amount1);
(uint256 _after0, uint256 _after1) = balances();
strategy.deposit();

_amount0 = _after0 - _bal0;
_amount1 = _after1 - _bal1;
```

An attacker could now call `deposit` in a situation in which adding a miniscule mount of token0 will cause the discontinuous jump in `balances()`, which would cause `_after0` to differ much more from `_bal0` than the miniscule amount of tokens used for the call. If the discontinuity were such that the value of `balances()` could be caused to jump *up* by increasing `total0` across the discontinuity, then this would mean the attacker would have to pay essentially nothing but get a decent amount of shares. Repeating this again and again, trading back across the discontinuity on the Uniswap pool in between calls to `deposit`, the attacker could gain a large number of shares and then cash out at the end for a large profit. However, crossing the discontinuity by increasing `total0` causes a jump *down*. This means that the caller to `deposit` would get *less* shares than they should get, so an attacker cannot profit from this directly.

With the amounts properly arranged, `_after0` could be made to be *smaller* than `_bal0`, however. This would cause the line `_amount0 = _after0 - _bal0;` to revert. An attacker might be able to front-run a deposit to engineer a value of `fees0` and a composition of the strategy's position that will cause such a revert. Regarding the contracts that are in scope themselves, this is only a griefing risk, but it cannot be ruled out that an attacker could profit from this in more complex interactions with other (third-party) contracts (as in, being able to make another contract's call to `deposit` revert opens up an attack vector on that other contract).

[2] Assuming we do not hit the issue with the locked profit, which we discuss below.
[3] We are using the intermediate value theorem here.

Furthermore, an attacker might be able to front-run a `deposit` transaction (to simplify, let us say one that only deposits token0) with a sandwich attack as follows to reduce the amount of shares the caller receives to `_minShares`. The transactions executed will be the following:

1. The attacker buys enough token0 for token1 so that `total0` and `fees0` are arranged in such a way that the following deposit will cross the discontinuity, with values such that the deposit will not revert but so that `_amount0 = _after0 - _bal0` will be small.
2. The original transaction. The value of `_amount0 = _after0 - _bal0` will be smaller than the amount of tokens paid by the caller due to the preceding transaction's manipulation.
3. The attacker sells their token0 for token1 to roll back their Uniswap pool manipulation.

Note that while an attacker can cause `_amount0 = _after0 - _bal0` to be miniscule, the deposit would then usually revert due to the number of shares minted being less than `_minShares`. If the caller were to set `_minShares=0`, then this attack could be used by an attacker that owns shares in the vault themselves in order to make a deposit act as a donation, from which they profit due to their shares being worth more afterwards.

There is a second discontinuity that is though more difficult to control by an attacker. The `lockedProfit` function contains a similar discontinuity. As `lockedProfit` is used to calculate `total0` in `balances()`, this discontinuity impacts `balances()` again.

```
function lockedProfit() public override view returns (uint256 locked0,
    uint256 locked1) {
    (uint256 balThis0, uint256 balThis1) = balancesOfThis();
    (uint256 balPool0, uint256 balPool1,,,,) = balancesOfPool();
    uint256 totalBal0 = balThis0 + balPool0;
    uint256 totalBal1 = balThis1 + balPool1;

    uint256 elapsed = block.timestamp - lastHarvest;
    uint256 remaining = elapsed < DURATION ? DURATION - elapsed : 0;

    // Make sure we don't lock more than we have.
    if (totalBal0 > totalLocked0) locked0 = totalLocked0 * remaining
    / DURATION;
    else locked0 = 0;

    if (totalBal1 > totalLocked1) locked1 = totalLocked1 * remaining
    / DURATION;
    else locked1 = 0;
}
```

Note that here `totalBal0` increasing across the discontinuity causes `locked0` to jump *up*, but as the value of `lockedProfits()` is subtracted in `balances()`, the resulting discontinuity in `balances()` still jumps *down*, as the one previously discussed.

Impact

Discontinuous behavior for balances in this kind of context can be very dangerous in general. In this case, the jump appears to be in the safer direction, and while there are a number of scenarios where an attacker might cause unintended behavior, we did not find a likely exploit in which the attacker can easily profit. However, more complicated attacks, or attacks involving other contracts that make use of and interact with the StrategyPassiveManagerUniswap or BeefyVaultConcLiq, cannot be ruled out. Changes to the vault contract that appear locally correct, assuming that `balances()` is continuous, could also potentially introduce serious vulnerabilities.

Recommendations

Consider removing the discontinuities in the two functions, for example by making the following changes. Firstly, for the `balances` function:

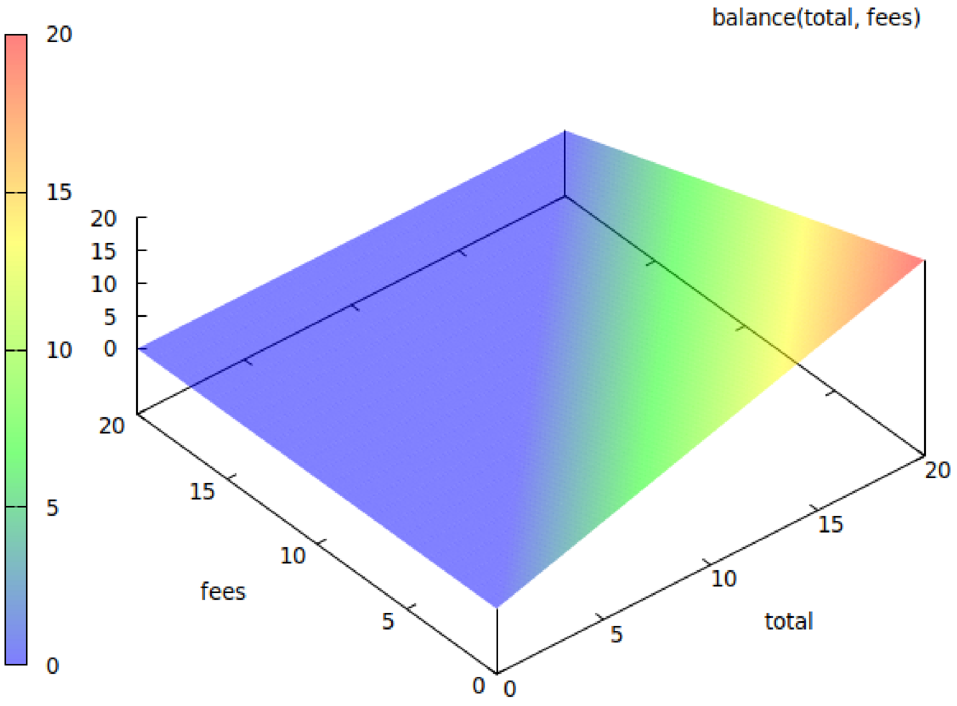
```
    function balances() public view returns (uint256 token0Bal, uint256
    token1Bal) {
        (uint256 thisBal0, uint256 thisBal1) = balancesOfThis();
        (uint256 poolBal0, uint256 poolBal1,,,,) = balancesOfPool();
        (uint256 locked0, uint256 locked1) = lockedProfit();

        uint256 total0 = thisBal0 + poolBal0 - locked0;
        uint256 total1 = thisBal1 + poolBal1 - locked1;
        uint256 unharvestedFees0 = fees0;
        uint256 unharvestedFees1 = fees1;

        // If pair is so imbalanced that we no longer have any enough tokens to
    pay fees, we set them to 0.
- if (unharvestedFees0 > total0) unharvestedFees0 = 0;
- if (unharvestedFees1 > total1) unharvestedFees1 = 0;
+ if (unharvestedFees0 > total0) unharvestedFees0 = total0;
+ if (unharvestedFees1 > total1) unharvestedFees1 = total1;

        // For token0 and token1 we return balance of this contract + balance
    of positions - locked profit - feesUnharvested.
        return (total0 - unharvestedFees0, total1 - unharvestedFees1);
    }
```

Here is a graph showing how the first return value of this changed function depends on `total0` and `fees0`, showing visually that the function is now continuous.



Secondly, the `lockedProfit` function:

```
    function lockedProfit() public override view returns (uint256 locked0,
    uint256 locked1) {
        (uint256 balThis0, uint256 balThis1) = balancesOfThis();
        (uint256 balPool0, uint256 balPool1,,,,) = balancesOfPool();
        uint256 totalBal0 = balThis0 + balPool0;
        uint256 totalBal1 = balThis1 + balPool1;

        uint256 elapsed = block.timestamp - lastHarvest;
        uint256 remaining = elapsed < DURATION ? DURATION - elapsed : 0;

        // Make sure we don't lock more than we have.
        if (totalBal0 > totalLocked0) locked0 = totalLocked0 * remaining /
    DURATION;
- else locked0 = 0;
+ else locked0 = totalBal0 * remaining / DURATION;
```

```
    if (totalBal1 > totalLocked1) locked1 = totalLocked1 * remaining /
  DURATION;
- else locked1 = 0;
+ else locked1 = totalBal1 * remaining / DURATION;
  }
```

In this case it may also make more semantic sense sense to also change the condition slightly to `totalBal0 > totalLocked0 * remaining / DURATION` instead of `totalBal0 > totalLocked0` and analogously for the second if.

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit `b949cf21` ↗.

3.4. Calm period not checked in all functions

Target	StrategyPassiveManagerUniswap		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

The `onlyCalmPeriods` modifier is a critical safety component of the strategy, as it is used to prevent deposits and harvests when the current price is not within 1% of the time-weighted average price (TWAP) over the last 60 seconds. This is to protect against price manipulation by an attacker during their transaction, possibly using flash loans or other means.

```
/// @notice Modifier to only allow deposit/harvest actions when current price
    is within 1% of twap.
modifier onlyCalmPeriods() {
    int24 tick = currentTick();
    int56 twapTick = twap();

    int56 upperBound = deviation + deviationThreshold;
    int56 lowerBound = deviationThreshold - deviation;

    // Calculate if greater than deviation % from twap and revert if it is.
    if (tick >= 0) {
        if (tick > twapTick * upperBound / deviationThreshold || tick <
    twapTick * lowerBound / deviationThreshold) revert NotCalm();
    } else if (tick < twapTick * upperBound / deviationThreshold || tick >
    twapTick * lowerBound / deviationThreshold) revert NotCalm();
    _;
}
```

One of the `harvest` functions as well as `withdraw` do not check whether the system is in a calm period or not. This means that users can still interact with the system, even though it is not safe to do so. This can lead to unexpected behavior and potential losses for the users.

```
function harvest() external {
    _harvest(tx.origin);
}

function withdraw(uint256 _amount0, uint256 _amount1) external {
    _onlyVault();
    // Liquidity has already been removed in beforeAction() so this is just a
    simple withdraw.
    if (_amount0 > 0) IERC20Metadata(lpToken0).safeTransfer(vault, _amount0);
    if (_amount1 > 0) IERC20Metadata(lpToken1).safeTransfer(vault, _amount1);

    // After we take what is needed we add it all back to our positions.
    if (!paused()) _addLiquidity();

    (uint256 bal0, uint256 bal1) = balances();

    // TVL Balances after withdraw
    emit Withdraw(bal0, bal1);
}
```

Impact

This issue can lead to unexpected slippage for the users as well as potential losses for the protocol as a whole.

Recommendations

We recommend ensuring that the `onlyCalmPeriods` modifier is used in all the external/public functions that can be called by anyone or by the vault. This will prevent users from interacting with the system when it is not safe to do so.

```
function harvest()
    external
    onlyCalmPeriods
{
    _harvest(tx.origin);
}

function withdraw(uint256 _amount0, uint256 _amount1)
    external
    onlyCalmPeriods
{
    _onlyVault();

    // Liquidity has already been removed in beforeAction() so this is just a
    simple withdraw.
    if (_amount0 > 0) IERC20Metadata(lpToken0).safeTransfer(vault, _amount0);
    if (_amount1 > 0) IERC20Metadata(lpToken1).safeTransfer(vault, _amount1);
    // After we take what is needed we add it all back to our positions.
    if (!paused()) _addLiquidity();

    (uint256 bal0, uint256 bal1) = balances();

    // TVL Balances after withdraw
    emit Withdraw(bal0, bal1);
}
```

Remediation

This issue has been acknowledged by Beefy, and fixes were implemented in the following commits:

- 9fbd3d43 ↗
- a7c127a2 ↗

3.5. Paused state is not checked

Target	StrategyPassiveManagerUniswap		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The strategy can be paused under certain periods that are considered to be dangerous for the system. This is usually done to protect the system from attacks or to prevent the system from making bad decisions.

However, not all functions perform checks on whether the system is paused or not. This means that users can still interact with the system, even though it is paused. This can lead to unexpected behavior and potential losses for the users.

The affected functions are essentially all the external/public functions that can be called by anyone or by the vault — for example, the `deposit` and `harvest` functions.

Impact

This issue can lead to unexpected behavior and potential losses for the users.

Recommendations

We recommend performing the necessary checks in all the external/public functions that can be called by anyone or by the vault. This will prevent users from interacting with the system when it is paused.

For example, the following modifier can be used in `deposit`:

```
function deposit()
    external
    onlyCalmPeriods
    whenNotPaused {
    _onlyVault();

    // Add All Liquidity
    _setTicks();
    _addLiquidity();
    (uint256 bal0, uint256 bal1) = balances();

    // TVL Balances after deposit
    emit Deposit(bal0, bal1);
}
```

Similarly, the same modifier can be used for the rest of the affected functions.

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit `9fbd3d43` ↗.

3.6. Withdrawals from vault might revert after donations

Target	BeefyVaultConcLiq		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The BeefyVaultConcLiq contract's `withdraw` function allows holders of the vault's shares to redeem them against the two tokens. The function calculates the amounts of the two tokens the users should be transferred, `_amount0` and `_amount1`, and obtains the balances `before0` and `before1` that the vault contract currently holds in these tokens. During normal operations, these values would be zero (see also 4.5. ↗). If the balance the vault holds is not enough to redeem the shares, the necessary tokens are withdrawn from the strategy contract. This is done starting from line 227:

```
if (before0 < _amount0 || before1 < _amount1) {

    uint _withdraw0 = _amount0 - before0;
    uint _withdraw1 = _amount1 - before1;

    strategy.withdraw(_withdraw0, _withdraw1);
```

This code will revert due to an arithmetic underflow if the branch is taken because one of the `before` values is smaller than the corresponding `_amount` value, but the other `before` value is larger than the corresponding `_amount` value (for example, if `before0 < _amount0` but `before1 < _amount1`).

Impact

Assume, as will be the case during normal operations, that the vault does not hold any of token0 or token1. An attacker could transfer some amount of one token, say x of token0, to the vault. Withdrawals of amounts of token0 smaller than x would then revert. This issue thus allows a denial-of-service attack on withdrawals, albeit one at a potentially significant cost to the attacker.

Recommendations

Consider replacing

```
uint _withdraw0 = _amount0 - before0;
```

by

```
uint _withdraw0;
if (_amount0 >= before0) _withdraw0 = _amount0 - before0;
```

to make `_withdraw0` zero when `_amount0 < before0`, and analogously for `_withdraw1`.

Alternatively, given that the vault should not hold tokens during normal operations (see also 4.5. ↗), the logic in `withdraw` could be simplified by always withdrawing the full amounts needed from the strategy contract and not using any tokens sent to the vault directly. This would mean, however, that any token0 or token1 sent to the vault accidentally would be stuck instead of being treated as a donation and used on withdrawals.

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit `8ff8e999` ↗.

3.7. The vault's `withdraw` function might not add liquidity again

Target	BeefyVaultConcLiq		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The strategy contract provides the tokens as liquidity in a Uniswap pool, thereby accumulating the relevant fees on the pool. For various actions by the strategy or vault contract, the liquidity is first withdrawn fully from the Uniswap pool, then added again. This is done to, for example, be able to reduce the amount locked in the Uniswap pool in order to be able to withdraw, or in order to have more precise accounting.

However, in the vault's `deposit` function, under some circumstances, it could happen that liquidity is first withdrawn but not added again:

```
function withdraw(uint256 _shares, uint256 _minAmount0, uint256 _minAmount1)
    public {
    // ...
    // Withdraw All Liquidity to Strat for Accounting.
    strategy.beforeAction();

    // ...
    if (before0 < _amount0 || before1 < _amount1) {

        uint _withdraw0 = _amount0 - before0;
        uint _withdraw1 = _amount1 - before1;

        strategy.withdraw(_withdraw0, _withdraw1);

        // ...
    }
    // ...
}
```

Towards the start of the `withdraw` function, `strategy.beforeAction()` is called, which will in particular remove liquidity. The only call afterwards in the function that will add liquidity again is `strategy.withdraw(_withdraw0, _withdraw1)`. However, this latter call is in a conditional branch, so if this branch is not executed, the removed liquidity would not be added again.

Impact

If it happens that `before0 < _amount0 || before1 < _amount1` is not true, then after this call to `withdraw`, all funds will held by the strategy contract, where they do not generate any profit, rather than being provided as liquidity in the Uniswap pools. The strategy contract will thus fail to realize profits it could otherwise have, until the next time a function is called that will cause liquidity to be added.

We note that in normal operations, `before0 < _amount0 || before1 < _amount1` should always be true.

Recommendations

Ensure that liquidity gets added again in `withdraw` after the call to `strategy.beforeAction()` no matter whether the conditional branch is taken. For example, considering that during normal operations the vault should not hold any tokens (see 4.5. ↗), it might be an option to replace lines 226 to 240 (inclusive) by the following line, also simplifying the logic significantly.

```
strategy.withdraw(_withdraw0, _withdraw1);
```

This solution would mean, however, that any token0 or token1 sent to the vault accidentally would be stuck instead of being treated as a donation and used on withdrawals.

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit `8ff8e999` ↗.

3.8. Vault's withdraw function might have slippage

Target	BeefyVaultConcLiq		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The vault contract takes two arguments `_minAmount0` and `_minAmount1` that are intended for the user to provide the minimum amount of tokens they expect to receive, with the call reverting if the amount they would actually receive would be lower than that due to slippage. In the function, the amount the user is to receive for their shares is first calculated as `_amount0` and `_amount1`, which are then checked to be at least `_minAmount0` and `_minAmount1`, with the function otherwise reverting. However, the amount actually transferred to the user later might be smaller than the value of `_amount0` and `_amount1` at this point, as the value of these variables might be reduced if the strategy failed to send a sufficient amount of tokens.

```
function withdraw(uint256 _shares, uint256 _minAmount0, uint256 _minAmount1)
    public {
    // ... _amount0 and _amount1 calculated here
    if (_amount0 < _minAmount0 || _amount1 < _minAmount1)
    revert TooMuchSlippage();

    (uint before0, uint before1) = available();
    if (before0 < _amount0 || before1 < _amount1) {
        // ...
        if (diff0 < _withdraw0) _amount0 = before0 + diff0; // _amount0
    changed to a possibly smaller value
        if (diff1 < _withdraw1) _amount1 = before1 + diff1;
    }

    (address token0, address token1) = wants();
    IERC20Upgradeable(token0).safeTransfer(msg.sender, _amount0); // the new
    value of _amount0 is used to transfer tokens to the caller, not the old one
    checked against _minAmount0
    IERC20Upgradeable(token1).safeTransfer(msg.sender, _amount1);

    emit Withdraw(msg.sender, _shares, _amount0, _amount1);
}
```

Impact

From the perspective of the vault's `withdraw` function, it could happen that the caller receives less tokens than the minimum amount they specified.

However, taking into account the actual implementation of the StrategyPassiveManagerUniswap contract's `withdraw` function, we see that the requested amount is always transferred exactly (`safeTransfer` will revert if the balance is insufficient).

```
function withdraw(uint256 _amount0, uint256 _amount1) external {
    _onlyVault();

    // Liquidity has already been removed in beforeAction() so this is just a
    simple withdraw.
    if (_amount0 > 0) IERC20Metadata(lpToken0).safeTransfer(vault, _amount0);
    if (_amount1 > 0) IERC20Metadata(lpToken1).safeTransfer(vault, _amount1);

    // ...
}
```

Hence, when the StrategyPassiveManagerUniswap contract is used as the strategy contract, this issue cannot occur, as `diff0 < _withdraw0` and `diff1 < _withdraw1` will never be true.

Recommendations

Consider checking the amount transferred to the caller against the minimum amount just before doing the actual transfer, thus taking into account any adjustments made before.

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit `8ff8e999` ↗.

3.9. Vault’s `deposit` function rounds to its disadvantage in divisor

Target	BeefyVaultConcLiq		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

To determine how many of the vault’s shares a caller depositing will receive, the vault’s `deposit` function calculates the value in token1 that the vault is currently worth (lines 178 and 179)

```
// How much of wants() do we have in token 1 equivalents;
uint256 token1EquivalentBalance = (_bal0 * price / PRECISION) + _bal1;
```

as well as the value in token1 that the caller’s deposit is worth (line 173):

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

Note that as all values are unsigned, the division by `PRECISION` will always round down. The number of shares the caller receives is then obtained by a calculation in which the value of `shares` as above is divided by `token1EquivalentBalance` (line 180).

```
shares = shares * _totalSupply / token1EquivalentBalance;
```

When calculating the amount of shares minted for a caller, it is best to always round in the vault’s favor, to avoid any manipulation that could result in an attacker receiving more shares than they should. For a quotient like here, this means the dividend should be rounded down and the divisor rounded up. Rounding down in the calculation of `token1EquivalentBalance` is instead to the caller’s advantage, as it reduces the divisor, hence increasing the quotient.

Impact

A caller might receive more shares than they should.

Recommendations

Consider rounding up in the calculation of `token1EquivalentBalance`, for example by replacing line 179 by the following line.

```
uint256 token1EquivalentBalance = (((_bal0 * price) + PRECISION - 1)
    / PRECISION) + _bal1;
```

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit `efb9e59e` ↗.

3.10. Uniswap mint reverts when exactly one of the second token is owed

Target	StrategyPassiveManagerUniswap		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When minting liquidity on a Uniswap V3 pool, the amount of tokens required are not transferred beforehand, but instead the Uniswap pool calls the `uniswapV3MintCallback` callback and passes it the amount of tokens that need to be transferred. This callback is implemented as follows.

```
function uniswapV3MintCallback(uint256 amount0, uint256 amount1,
    bytes memory /*data*/) external {
    if (msg.sender != pool) revert NotPool();
    if (!minting) revert InvalidEntry();

    if (amount0 > 0) IERC20Metadata(lpToken0).safeTransfer(pool, amount0);
    if (amount1 > 1) IERC20Metadata(lpToken1).safeTransfer(pool, amount1);
    minting = false;
}
```

In the case where `amount1 = 1`, this would not transfer any token1 to the Uniswap pool, which would then make the minting of liquidity revert.

Impact

Minting of liquidity by `_addLiquidity` can revert in a very rare edge case.

Recommendations

```
function uniswapV3MintCallback(uint256 amount0, uint256 amount1,
    bytes memory /*data*/) external {
    if (msg.sender != pool) revert NotPool();
    if (!minting) revert InvalidEntry();

    if (amount0 > 0) IERC20Metadata(lpToken0).safeTransfer(pool, amount0);
     if (amount1 > 1) IERC20Metadata(lpToken1).safeTransfer(pool,
         amount1);
     if (amount1 > 0) IERC20Metadata(lpToken1).safeTransfer(pool,
         amount1);
    minting = false;
}
```

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit 78b20f76 ↗.

3.11. Admins might donate fees on `panic` and `setPositionWidth`

Target	StrategyPassiveManagerUniswap		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The strategy contract's `_removeLiquidity` function calls the Uniswap pool's `burn` function first to burn liquidity, followed by `collect` to collect any tokens owed by the pool to the strategy. This will also pay out any owed fees. For this reason, calls to `_removeLiquidity` are usually preceded by a call to `_claimEarnings`, which will use `collect` to collect any outstanding fees earned on Uniswap and account for them properly, so that, for example, any Beefy fees on them can be deducted correctly.

The functions `setPositionWidth` and `panic`, however, call `_removeLiquidity` without a preceding call to `_claimEarnings`, which will thus in effect donate any outstanding fees on the Uniswap pool to the vault's shareholders without deducing any fees. These two functions can though only be called by the owner and manager, respectively.

Impact

During some rare administrative tasks, fees might not be deduced from Uniswap fees earned.

Recommendations

Consider calling `_claimEarnings` before `_removeLiquidity` in the two mentioned functions. Alternatively, document this behavior and consider calling `harvest` before calls to the two functions to ensure fees have been accounted for.

Remediation

This issue has been acknowledged by Beefy, and a fix was implemented in commit `2c2db9ae` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Impact of manipulations of the Uniswap pool price

In this section, we discuss the impact of manipulations of the Uniswap pool price by a possible attacker within a single transaction, that is without manipulating the time-weighted average price. Let us denote the two tokens by X and Y. We assume that the time-weighted average price is P (this is in units of $\frac{Y}{X}$), and in the positive direction, the `onlyCalmPeriods` modifier checks that the current price at the time of invocation p satisfies $p \leq (1+d) \cdot P$ for some constant $d \geq 0$. We will sketch an attack scenario in which the attacker attempts to use price manipulations in order to steal value from the vault/strategy contracts, and we will attempt to estimate (making some simplifications along the way) under what conditions, in particular what value of d, the attacker can make a profit. We assume the price before the attacker begins is P and that the value of the strategy contract's position is V (in units of Y).

The attacker takes the following actions with funds of tokens X and Y potentially obtained from a flash loan, and paid back at the end. We will expand on the precise amounts later.

1. They buy a large amount of X in the Uniswap pool with their Y, moving the price to $p = (1+d) \cdot P$.
2. They deposit some tokens X into the vault.
3. They sell a large amount of X in the Uniswap pool, moving the price back to P.
4. They withdraw their share of the vault.

Step 1: Manipulating the price

To simplify our calculations, we assume that the liquidity provided by the strategy to the Uniswap pool is centered around the tick corresponding to the price P, with a very small width, so that we can ignore the price change while trading within that position for the purposes of our estimate. This position, valued V in total, is composed of X and Y so that half of the value is in each of the two tokens, so it consists of $a_Y = \frac{V}{2}$ tokens Y and $a_X = \frac{V}{2 \cdot P}$ tokens X.

The attackers buying converts the strategy's position entirely to token Y, so the attacker will pay a_Y in Y for this and receive a_X in X. Beyond that, let us assume that the rest of the liquidity in the Uniswap pool amounts to having to pay b_Y in Y and obtain b_X in X to move the price to $p = (1+d) \cdot P$.

The above was without fees. Let us assume the fee rate for the Uniswap pool is γ, for example $\gamma = 1.0005$ (i.e., 0.05%). Then after Step 1, the attacker has paid $(a_Y + b_Y) \cdot (1-\gamma)^{-1}$ of token Y and received $a_X + b_X$ of token X.

Step 2: Depositing tokens

The attacker now deposits c_X tokens X. Note that due to this step, we must restrict our price manipulation in such a way as to pass the `onlyCalmPeriods` check.

The first thing that will happen is that the strategy removes its liquidity. Ignoring the minor amount of fees, this results in the entire value V being returned in the form of token Y. The price of the c_X tokens X contributed by the attacker is $c_X \cdot (1+d) \cdot P$. If the total amount of shares before was T, then the attacker thus receives $\frac{T \cdot c_X \cdot (1+d) \cdot P}{V}$.

The strategy contract will then provide liquidity to the Uniswap pool around the current price, providing c_X of token X and V of token Y.

Step 3: Returning the price to normal

To trade back the price in the Uniswap pool to where it was before, the attacker first needs to trade through the strategy contract's position, which means converting the tokens Y to X, at a price of $(1+d) \cdot P$ (we again ignore the price changes due to the width of the position). As the strategy contract's position consisted of V of token Y, the attacker must pay $d_X = \frac{V}{(1+d) \cdot P}$ of token X for this without fees, and they receive V of token Y in return.

After that, the attacker has to pay b_X in X and receive b_Y in Y, reverting the trade regarding the remaining liquidity in the pool between the prices.

Taking into account Uniswap fees, in total the attacker has to pay $(d_X + b_X) \cdot (1-\gamma)^{-1}$ in X and receives $V + b_Y$ in token Y.

Step 4: Withdrawing the shares

When withdrawing the shares, the strategy will first withdraw its liquidity, which amounts to $c_X + d_X$ of token X. We can rewrite this as follows.

$$\begin{aligned} c_X + d_X &= c_X + \frac{V}{(1+d) \cdot P} \\ &= \frac{c_X \cdot (1+d) \cdot P + V}{(1+d) \cdot P} \end{aligned}$$

The attacker holds $\frac{T \cdot c_X \cdot (1+d) \cdot P}{V}$ of a total of $T + \frac{T \cdot c_X \cdot (1+d) \cdot P}{V}$ shares, and so they receive the following amount of tokens X. They thus receive the following amount of tokens X.

$$\begin{aligned} e_X &= \frac{T \cdot c_X \cdot (1+d) \cdot P}{V} \cdot \left(T + \frac{T \cdot c_X \cdot (1+d) \cdot P}{V}\right)^{-1} \cdot (c_X + d_X) \\ &= \frac{T \cdot c_X \cdot (1+d) \cdot P}{V} \cdot \left(\frac{T \cdot V + T \cdot c_X \cdot (1+d) \cdot P}{V}\right)^{-1} \cdot (c_X + d_X) \\ &= \frac{T \cdot c_X \cdot (1+d) \cdot P}{T \cdot V + T \cdot c_X \cdot (1+d) \cdot P} \cdot (c_X + d_X) \\ &= \frac{c_X \cdot (1+d) \cdot P}{V + c_X \cdot (1+d) \cdot P} \cdot \frac{c_X \cdot (1+d) \cdot P + V}{(1+d) \cdot P} \\ &= c_X \end{aligned}$$

End result for the vault

At the end, the vault is left with only tokens X. Let us calculate their worth in terms of token Y in order to compare with the original worth of V.

$$\begin{aligned} V_{after} &= ((c_X + d_X) - e_X) \cdot P \\ &= c_X + \left(\frac{V}{(1+d) \cdot P} - c_X\right) \cdot P \\ &= \frac{V}{(1+d) \cdot P} \cdot P \\ &= \frac{V}{(1+d)} \end{aligned}$$

According to the above formula, the vault/strategy contracts lose value, with loss increasing the bigger $d > 0$ is. For very small d, this loss might be offset by the fees collected during the attack. For the first step, the fees collected will be about $\frac{V}{2} \cdot \frac{\gamma}{1-\gamma}$ in token Y. The fees collected when trading back will be about $\frac{V}{(1+d) \cdot P} \cdot \frac{\gamma}{1-\gamma}$ in token X. The total worth of the fees is thus, in Y,

$$\begin{aligned} V_{fees} &= \frac{V}{2} \cdot \frac{\gamma}{1-\gamma} + \frac{V}{(1+d) \cdot P} \cdot \frac{\gamma}{1-\gamma} \cdot P \\ &= V \cdot \left(\frac{1}{2} + \frac{1}{1+d}\right) \cdot \frac{\gamma}{1-\gamma} \end{aligned}$$

We can easily check that these results are plausible: the strategy traded its entire value from one token to the other at the unfavorable price of $(1+d) \cdot P$ instead of P, so V_{after} is as one would expect. Similarly, fees collected come from trading half of the liquidity at the start, and then the entire liquidity, but at the unfavorable price, so $\frac{1}{2} + \frac{1}{1-d}$ is as expected as well.

Conditions under which the vault makes a loss

For what d will the fees we just estimated offset the loss? For the vault not losing value on this attack, we obtain the following, assuming $\gamma < 2/3$:

$$\begin{aligned}
& \frac{V}{(1+d)} + V \cdot \left(\frac{1}{2} + \frac{1}{1+d}\right) \cdot \frac{\gamma}{1-\gamma} \geq V \\
\Longleftrightarrow\ & \frac{1}{(1+d)} + \left(\frac{1}{2} + \frac{1}{1+d}\right) \cdot \frac{\gamma}{1-\gamma} \geq 1 \\
\Longleftrightarrow\ & \frac{1+\frac{\gamma}{1-\gamma}}{(1+d)} + \frac{\gamma}{2-2\gamma} \geq 1 \\
\Longleftrightarrow\ & \frac{1+\frac{\gamma}{1-\gamma}}{(1+d)} \geq 1 - \frac{\gamma}{2-2\gamma} \\
\Longleftrightarrow\ & \frac{1+\frac{\gamma}{1-\gamma}}{1-\frac{\gamma}{2-2\gamma}} \geq 1+d \\
\Longleftrightarrow\ & \frac{1+\frac{\gamma}{1-\gamma}}{1-\frac{\gamma}{2-2\gamma}} - 1 \geq d
\end{aligned}$$

Let us calculate this for a concrete value for γ. Uniswap V3 pools can have different fees. A lower fee will be worse for the vault/strategy here, so to be conservative, let us consider a low fee of 0.01%, for which there for example exists a DAI/USDC pool with significant activity. Using this script,

```
gamma = 0.0001
num = 1 + (gamma / (1-gamma))
denom =  1 - (gamma / 2*(1 - gamma))
result = num/denom - 1
print(result)
```

we obtain that we must have roughly $d < 0.00015$ to not make a loss, amounting to a relative price change from the time-weighted average price of at most 0.015%. It thus might not always be feasible to prevent this kind of attack by choosing d in such a way that the vault/strategy will never make a loss; prevention must hinge on the attacker being unable to make a profit instead (due to having to trade through the rest of the liquidity providers' liquidity as well).

End result for the attacker

Let us calculate the net amount of token X and Y the attacker gained. We obtain the following.

$$\begin{aligned}\Delta_X &= a_X + b_X - c_X - (d_X + b_X)\cdot(1-\gamma)^{-1} + c_X\\ &= a_X + b_X - (d_X + b_X)\cdot(1-\gamma)^{-1}\\ &= \frac{V}{2\cdot P} + b_X - \left(\frac{V}{(1+d)\cdot P} + b_X\right)\cdot(1-\gamma)^{-1}\\ &= \frac{V}{2\cdot P} - \frac{V}{(1+d)\cdot P\cdot(1-\gamma)} - b_X\cdot\frac{\gamma}{1-\gamma}\\ &= \frac{V}{P}\cdot\left(\frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)}\right) - b_X\cdot\frac{\gamma}{1-\gamma}\end{aligned}$$

$$\begin{aligned}\Delta_Y &= -(a_Y + b_Y)\cdot(1-\gamma)^{-1} + V + b_Y\\ &= -\left(\frac{V}{2} + b_Y\right)\cdot(1-\gamma)^{-1} + V + b_Y\\ &= \frac{V\cdot(1-2\gamma)}{2\cdot(1-\gamma)} - b_Y\cdot\frac{\gamma}{1-\gamma}\end{aligned}$$

Let us convert the total value to token Y. To simplify, we assume that the liquidity in the Uniswap pool was concentrated around the price P, so that we can estimate $b_Y = b_X\cdot P$. Then we obtain

$$\begin{aligned}\Delta_X\cdot P + \Delta_Y &= V\cdot\left(\frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)}\right) - b_Y\cdot\frac{\gamma}{1-\gamma} + \frac{V\cdot(1-2\gamma)}{2\cdot(1-\gamma)} - b_Y\cdot\frac{\gamma}{1-\gamma}\\ &= V\cdot\left(\frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)} + \frac{1-2\gamma}{2\cdot(1-\gamma)}\right) - b_Y\cdot\frac{2\gamma}{1-\gamma}\end{aligned}$$

We can check plausibility of this formula: the $V\cdot\left(\frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)} + \frac{1-2\gamma}{2\cdot(1-\gamma)}\right)$ part comes from the attacker trading with the strategy's positions. If we set $\gamma = 0$, this would be $V\cdot\left(1 - \frac{1}{(1+d)}\right)$, which corresponds to buying the strategy's entire balance while only needing to pay $\frac{1}{1+d}$ of the fair price for it, which is exactly what the attack is doing, so this is exactly as we would expect. The occurence of γ lowers the value due to losses due to fees. The $b_Y\cdot\frac{2\gamma}{1-\gamma}$ part is then from the fees required to trade through the other liquidity provider's liquidity twice.

Conditions under which the attack is profitable

Given a certain ratio between V and b_Y, say $r = V/b_Y$, what values of d allow the attacker to make a profit? We obtain

$$
\begin{aligned}
& V \cdot \left(\frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)} + \frac{1-2\gamma}{2\cdot(1-\gamma)}\right) - b_Y \cdot \frac{2\gamma}{1-\gamma} \geq 0 \\
\iff & r \cdot b_Y \cdot \left(\frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)} + \frac{1-2\gamma}{2\cdot(1-\gamma)}\right) - b_Y \cdot \frac{2\gamma}{1-\gamma} \geq 0 \\
\iff & r \cdot \left(\frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)} + \frac{1-2\gamma}{2\cdot(1-\gamma)}\right) - \frac{2\gamma}{1-\gamma} \geq 0 \\
\iff & \frac{1}{2} - \frac{1}{(1+d)\cdot(1-\gamma)} + \frac{1-2\gamma}{2\cdot(1-\gamma)} \geq r^{-1} \cdot \frac{2\gamma}{1-\gamma} \\
\iff & \frac{1}{2} + \frac{1-2\gamma}{2\cdot(1-\gamma)} - r^{-1} \cdot \frac{2\gamma}{1-\gamma} \geq \frac{1}{(1+d)\cdot(1-\gamma)} \\
\iff & (1-\gamma) \cdot \left(\frac{1}{2} + \frac{1-2\gamma}{2\cdot(1-\gamma)} - r^{-1} \cdot \frac{2\gamma}{1-\gamma}\right) \geq \frac{1}{1+d}
\end{aligned}
$$

There are two cases here. If the left-hand side is smaller than or equal to zero, then the attacker will not make a profit no matter what d is. Otherwise we get

$$
d \geq \left((1-\gamma) \cdot \left(\frac{1}{2} + \frac{1-2\gamma}{2\cdot(1-\gamma)} - r^{-1} \cdot \frac{2\gamma}{1-\gamma}\right)\right)^{-1} - 1
$$

Examples for profitability

Using the above formulas, we arrive at the following table for when the attack will be profitable.[4]

[4] Note that this is under the simplifying assumptions we made in our calculation, such as price not changing significantly within the strategy's position.

γ	r	Requirement for d
0.0001	1.0	$d \geq 0.04\%$
0.0001	0.1	$d \geq 0.22\%$
0.0001	0.01	$d \geq 2.06\%$
0.0001	0.001	$d \geq 25.02\%$
0.001	1.0	$d \geq 0.35\%$
0.001	0.1	$d \geq 2.20\%$
0.001	0.01	$d \geq 25.23\%$
0.001	0.001	Not possible
0.01	1.0	$d \geq 3.63\%$
0.01	0.1	$d \geq 27.39\%$
0.01	0.01	Not possible
0.01	0.001	Not possible

We can see that, for example, for a reasonable ratio of 1%,[5] and a fee of 0.01% ($\gamma = 0.0001$), a value of $d \geq 2.06\%$ suffices to make the attack profitable.

Code used to generate the profitability table

The following Sage code was used to generate the table above.

```
def lhs(gamma, r):
  a = 1/2
  b = (1 - 2*gamma) / (2 - 2*gamma)
  c = (r**(-1)) * ((2*gamma) / (1 - gamma))
  inner = a + b - c
  outer = (1 - gamma) * inner
  return outer

def calc(gamma, r):
  return (lhs(gamma, r)**(-1)) - 1

print('| `l!$\gamma$` | `l!$r$` | requirement for `l!$d$` |')
print('| --- | --- | --- |')
for gamma in (0.0001, 0.001, 0.01):
  for r in (1, 0.1, 0.01, 0.001):
    if lhs(gamma, r) <= 0:
      result = 'not possible'
    else:
      result = f'`l!$d \geq {float(calc(gamma, r)*100):.2f}$`%'
    print(f'| `l!${float(gamma)}$` | `l!${float(r)}$` | {result}  |')
```

Disclaimer

The above estimates are a best effort based on simplifying assumptions (such as ticks being continuous real numbers rather than discrete, tick spacing and position width infinitesimal, etc.), so the numbers obtained may not be precise boundaries that are safe.

4.2. Centralization risk

There are two types of privileged accounts for the StrategyPassiveManagerUniswap contract:

[5] Note that this does take into account all of the liquidity that the strategy provides but only part of the liquidity other liquidity providers provide, so the ratio taking all into account would usually be about half of this only — and less than that if liquidity also exists further out from the time-weighted average price than the range we manipulate the price in.

- The owner
- The manager

The manager can call the `panic` and `unpause` functions, which essentially pause the system, remove the liquidity and the allowances, or unpause the system and add the allowances and the liquidity.

The owner can do everything the manager can do and also set the paths for the token0/token1 to native swaps, set the allowance of deviation from the TWAP, and set the position width.

Because the owner can essentially set the deviation from the TWAP and the position width, they could theoretically manipulate the system to their advantage. This is an important consideration for the security of the system, as it could lead to a loss of funds for the users of the system.

Special consideration should be given to the `inCaseTokensGetStuck` function, which allows the owner to withdraw tokens from the Vault contract. At the time of the audit, the funds do not persist in the Vault contract, as they're transferred straight away to the underlying strategies during any action, so it does not pose a direct security threat. This persistency of funds might change in the future, if the team was to use the Vault to actually store funds. In that case, the `inCaseTokensGetStuck` function would allow the owner to withdraw all funds from the Vault, posing a significant risk to all the users of the protocol.

The Beefy team considers improving decentralization in the future; however, it is important to note that the current implementation is not fully decentralized.

4.3. Initializers are not called for cloned contracts

The factories' contracts offer the ability to clone contracts, which is a useful feature for the overall composability of the system. However, the initializers of the cloned contracts are not called on the spot, which may lead to unusable contracts, or even to security issues, should a malicious user call `initialize()` themselves.

- The BeefyVaultConcLiqFactory has `cloneVault()`, where the `BeefyVaultConcLiq` vault is cloned. However, the `initialize()` function is not called after the cloning, which means that the vault is not properly initialized.
- The StrategyFactory has `createStrategy()`, where the `BeefyStrategy` strategy is cloned. However, the `initialize()` function is not called after the cloning, which means that the strategy is not properly initialized.

We recommend ensuring that the initializers are called after the cloning of the contracts but in the same transaction. This will ensure that the contracts are properly initialized and usable.

4.4. Test suite

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

It is important to test the invariants required for ensuring security and also verify mathematical properties as specified in the white paper and documentation. Additionally, testing under various market conditions is essential for ensuring that the system operates as intended.

Even though the current testing suite touches on most functionalities, we recommend expanding it to include more edge cases, especially around the specific tokens that are used as well as the AMM (in this case, the UniswapV3 pools). This is especially important as the strategies are essentially in a symbiotic relationship with the UniswapV3 pools, and the behavior of the system is highly dependent on the behavior of the pools, which can be unpredictable under chaotic market conditions.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

4.5. The vault's `withdraw` function could be simplified

The BeefyVaultConcLiq contract contains the following piece of code from line 226 to 240:

```
(uint before0, uint before1) = available();
if (before0 < _amount0 || before1 < _amount1) {

    uint _withdraw0 = _amount0 - before0;
    uint _withdraw1 = _amount1 - before1;

    strategy.withdraw(_withdraw0, _withdraw1);

    (uint after0, uint after1) = available();
    uint diff0 = after0 - before0;
    uint diff1 = after1 - before1;

    if (diff0 < _withdraw0) _amount0 = before0 + diff0;
    if (diff1 < _withdraw1) _amount1 = before1 + diff1;
}
```

This code snippet could be simplified.

As diff0 and diff1 are not used after the the two ifs directly after their definition, their definition could be removed and the two ifs near the end replaced by the equivalent following lines.

```
if(_after0 < _amount0) _amount0 = after0;
if(_after1 < _amount1) _amount1 = after1;
```

If the strategy contract used is StrategyPassiveManagerUniswap, then we will always have diff0 = _withdraw0 and diff1 = _withdraw1 (see Finding 3.8. ↗), so these lines could be removed completely as well.

More generally, during normal operation, the vault only obtains tokens temporarily exactly due to the strategy.withdraw(_withdraw0, _withdraw1) call, and those tokens are then (assuming the vault's previous balance was zero) directly transferred to the caller. Outside of calls to the vault's withdraw function, the vault would thus usually not hold any balance of token0 or token1 and could only do so if sent tokens directly. Given this, one could consider to simplify the entire piece of code quoted above by replacing it with

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

During normal operations with the StrategyPassiveManagerUniswap contract as strategy contract, this would make the withdraw function functionally equivalent to the previous version, with the exception of Findings 3.6. ↗ and 3.7. ↗ having been fixed. There would be one noticeable difference in functionality: should users, perhaps accidentally, send tokens to the vault, then the previous version would treat those like a donation. This replacement would instead not make use of these tokens when paying out withdrawals. These tokens would instead be stuck. In order to avoid such tokens influencing the price of vault shares, one should then not count them anymore in balances().

4.6. Observation cardinality bigger than needed

The StrategyPassiveManagerUniswap contract makes use of the time-weighted average tick from the Uniswap pool over the last 60 seconds:

```
/**
 * @notice The twap of the last minute from the pool.
 * @return twapTick The twap of the last minute from the pool.
*/
function twap() public view returns (int56 twapTick) {
    uint32[] memory secondsAgo = new uint32[](2);
    secondsAgo[0] = 60;
    secondsAgo[1] = 0;

    (int56[] memory tickCuml,) = IUniswapV3Pool(pool).observe(secondsAgo);
    twapTick = (tickCuml[1] - tickCuml[0]) / 60;
}
```

When Uniswap pools get created, they start out with only a single observation checkpoint for calculating time-weighted average tick. However, by calling `increaseObservationCardinalityNext`, it is possible to increase the number of observation checkpoints. This is what the strategy contract's initializer does:

```
// Since Uniswap V3 pools init with a 1 observation cardinality we need to
    increase it to 60 to get more accurate twap, if not already increased.
(,,,uint16 cardinality,,,) = IUniswapV3Pool(pool).slot0();
if (cardinality < 60)
    IUniswapV3Pool(pool).increaseObservationCardinalityNext(60);
```

The Uniswap pool only stores at most one checkpoint on each block, however. So to cover t seconds, the observation cardinality needed is about $\lceil \frac{t}{\tau} \rceil + 1$, where τ is the time passing between two blocks. For the Ethereum Mainnet, τ is about 12 seconds, so no more than six observation checkpoints would be expected to be needed for a 60-second period under normal conditions. On other blockchains with less time between blocks or more variance in block timings, more would be needed.

4.7. Positions might change more than expected due to tick spacing

Positions can only be initialized at ticks divisible by `tickSpacing`, which is a value set by Uniswap. In the StrategyPassiveManagerUniswap contract, when setting positions, the current tick is thus rounded down to the next value divisible by `tickSpacing` to obtain the midpoint of the position:

```
/// @dev Calc base ticks depending on base threshold and tickspacing
function baseTicks(
    int24 currentTick,
    int24 baseThreshold,
    int24 tickSpacing
) internal pure returns (int24 tickLower, int24 tickUpper) {
    int24 tickFloor = floor(currentTick, tickSpacing);

    tickLower = tickFloor - baseThreshold;
    tickUpper = tickFloor + baseThreshold;
}
```

This means that the tick moving across a number divisible by `tickSpacing` can cause the position to move a full `tickSpacing`, which may be much larger than the tick change. For example, with `tickSpacing` being 10 and the position previously having been set at a tick of 19, the tick increasing by only 1 to 20, will cause the position range to move up by 10.

This effect should be taken in mind when considering the settings for `_onlyCalmPeriods`.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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. Module: BeefyVaultConcLiqFactory.sol

Function: `cloneVault()`

Creates a clone of a Beefy Conc Liq Vault.

Branches and code coverage

Intended branches

- Ensure initialize is called on the new vault. Currently not implemented.
  - ☐ Test coverage
- Clone the new vault.
  - ☑ Test coverage
- Emit the `ProxyCreated` event.
  - ☑ Test coverage

5.2. Module: BeefyVaultConcLiq.sol

Function: `depositAll(uint256 _minShares)`

Allows user to deposit in the vault their entire balances of token0 and token1.

Inputs

- `_minShares`
  - **Control**: Fully controlled by the user.
  - **Constraints**: none.
  - **Impact**: The minimum amount of shares that the user wants to receive with slippage.

Branches and code coverage

Intended branches

- Call `deposit` with the entire balance of each tokens for the user.
  - ☑ Test coverage

Negative behavior

- Should not allow the user to deposit more than they have. This is enforced through the deposit function.
  - ☑ Negative test

Function: `deposit(uint256 _amount0, uint256 _amount1, uint256 _minShares)`

Allows user to deposit in the vault and receive shares.

Inputs

- `_amount0`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None. Basically checked that `msg.sender` affords the deposit.
  - **Impact**: The amount of token0 to deposit.
- `_amount1`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None. Basically checked that `msg.sender` affords the deposit.
  - **Impact**: The amount of token1 to deposit.
- `_minShares`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None. Checked that the user receives enough shares.
  - **Impact**: The minimum amount of shares that the user wants to receive with slippage.

Branches and code coverage

Intended branches

- Should transfer the `_amount0` and `_amount1` from the user to the strategy.
  - ☑ Test coverage
- Should mint the user the correct amount of shares.
  - ☑ Test coverage
- Ensure that the user receives enough shares (i.e., `shares > _minShares`).
  - ☑ Test coverage
- Calculate the correct amount of shares to mint based on the amount of tokens deposited.
  - ☑ Test coverage
- Should mint some minimum amount of shares on the first-ever deposit. This is against

the first-depositor bug.
  ☑ Test coverage

Negative behavior

- Should not allow the user to deposit zero tokens. Currently not enforced.
  - □ Negative test
- Should not allow the user to deposit more than they have.
  - ☑ Negative test
- Should not allow the user to deposit someone else's tokens.
  - ☑ Negative test
- Should not allow the user to deposit inflationary tokens.
  - ☑ Negative test

Function: `inCaseTokensGetStuck(address _token)`

Function to transfer stuck tokens out of the contract.

Inputs

- `_token`
  - **Control**: Fully controlled by owner.
  - **Constraints**: Checked that it is not token0 or token1.
  - **Impact**: The token to be transferred out of the contract.

Branches and code coverage

Intended branches

- Ensure that the token is neither token0 nor token1.
  - ☑ Test coverage
- Transfer the token to the owner.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  - ☑ Negative test

Function: `withdrawAll(uint256 _minAmount0, uint256 _minAmount1)`

Allows the user to withdraw their entire balance from the vault.

Inputs

- `_minAmount0`
    - **Control**: Fully controlled by the user.
    - **Constraints**: None.
    - **Impact**: The minimum amount of token0 that the user wants to receive with slippage.
- `_minAmount1`
    - **Control**: Fully controlled by the user.
    - **Constraints**: None.
    - **Impact**: The minimum amount of token1 that the user wants to receive with slippage.

Branches and code coverage

Intended branches

- Call the withdraw function with the entire balance of the user.
    - ☑ Test coverage

Negative behavior

- Should not allow the user to withdraw more than they have. This is enforced through the withdraw function.
    - ☑ Negative test
- Should not allow the user to withdraw someone else's shares.
    - ☑ Negative test

Function: `withdraw(uint256 _shares, uint256 _minAmount0, uint256 _minAmount1)`

Allows users to withdraw their shares from the vault.

Inputs

- `_shares`
    - **Control**: Fully controlled by the user.
    - **Constraints**: Checked that the user has enough shares to withdraw.
    - **Impact**: The number of shares to be withdrawn.
- `_minAmount0`
    - **Control**: The minimum amount of token0 that the user wants to receive with slippage.
    - **Constraints**: Checked that the resulting amount is greater than the minimum amount.

- **Impact**: The minimum amount of token0 that the user wants to receive with slippage.
- `_minAmount1`
  - **Control**: The minimum amount of token1 that the user wants to receive with slippage.
  - **Constraints**: Checked that the resulting amount is greater than the minimum amount.
  - **Impact**: The minimum amount of token1 that the user wants to receive with slippage.

Branches and code coverage

Intended branches

- Assumes that the position of the vault does have enough liquidity to withdraw the shares.
  - ☑ Test coverage
- Calculate the amount of tokens needed to be withdrawn for the burnt shares.
  - ☑ Test coverage
- Should burn the shares from `msg.sender`.
  - ☑ Test coverage
- Should transfer at least `_minAmount0` of `token0` and `_minAmount1` of `token1` to `msg.sender`.
  - ☑ Test coverage

Negative behavior

- Should not allow the user to withdraw more shares than they have. This is enforced through burn, as the user must have the shares to burn them.
  - ☑ Negative test

5.3. Module: StrategyFactory.sol

Function: `addStrategy(string _strategyName, address _implementation)`

Allows adding a new strategy to the factory.

Inputs

- `_strategyName`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Check if the strategy name already exists.
  - **Impact**: The name of the strategy to add.
- `_implementation`
  - **Control**: Fully controlled by the caller.

- **Constraints**: None.
- **Impact**: The implementation address.

Branches and code coverage

Intended branches

- Ensure that the `_implementation` is a valid contract address.
  - ☐ Test coverage
- Creates a new upgradable beacon instance for the given strategy name.
  - ☑ Test coverage

Negative behavior

- Should not allow overwriting an existing strategy.
  - ☑ Negative test
- Should not be callable by anyone other than the manager.
  - ☑ Negative test

Function: `createStrategy(string _strategyName)`

Allows creating a new Beefy Strategy as a proxy of the template instance.

Inputs

- `_strategyName`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None. Should ensure that `strategyName` exists in the `instances` mapping.
  - **Impact**: The name of the strategy to copy.

Branches and code coverage

Intended branches

- Should call the initializer function on the new strategy. Currently not performed.
  - ☐ Test coverage
- Create a new `BeaconProxy` instance.
  - ☑ Test coverage
- Push the new strategy to the strategies array.
  - ☑ Test coverage

Negative behavior

- Should not allow creating a new strategy if the strategy name does not exist in the `instances` mapping. Currently not checked.
  - ☐ Negative test

Function: `upgradeTo(string _strategyName, address _newImplementation)`

Allows upgrading the implementation of a strategy.

Inputs

- `_strategyName`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None. Should ensure that `strategyName` exists in the `instances` mapping.
  - **Impact**: The name of the strategy to upgrade.
- `_newImplementation`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The new implementation address.

Branches and code coverage

Intended branches

- Should check that the `_newImplementation` is a valid contract address.
  - ☐ Test coverage
- Should check that an instance of the `_strategyName` exists.
  - ☐ Test coverage
- Upgrades the instance to the new implementation.
  - ☑ Test coverage

Negative behavior

- Assumes that the current implementation is not the same as the new implementation.
  - ☐ Negative test
- Should not be callable by anyone other than the owner.
  - ☑ Negative test

5.4. Module: StrategyPassiveManagerUniswap.sol

Function: `beforeAction()`

Function to be called by the vault before depositing or withdrawing to remove liquidity and harvest fees for accounting purposes.

Branches and code coverage

Intended branches

- Claim the earnings.
  - ☑ Test coverage
- Remove the liquidity.
  - ☑ Test coverage
- Assumes the liquidity will be added back after the action.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the vault.
  - ☑ Negative test

Function: `deposit()`

Allows vault to deposit all liquidity back to their positions.

Branches and code coverage

Intended branches

- Should update the ticks.
  - ☑ Test coverage
- Should add liquidity.
  - ☑ Test coverage
- Trusts the vault in that it assumes the funds have been transferred before calling this function.
  - ☑ Test coverage

Negative behavior

- Should only be callable by the vault.
  - ☑ Negative test
- Should not allow calling this if the strategy is paused.
  - □ Negative test

Function: `harvest()`

Harvest fees from the pool, charge fees for Beefy, then readjust positions.

Branches and code coverage

Intended branches

- Reset the `fees0` counter.
  - □ Test coverage
- Reset the `fees1` counter.
  - □ Test coverage
- Should forward part of the fees to `tx.origin`.
  - ☑ Test coverage
- Set the `totalLocked0` counter.
  - ☑ Test coverage
- Set the `totalLocked1` counter.
  - ☑ Test coverage
- Set the `lastHarvest` to the current block timestamp.
  - ☑ Test coverage
- Claim fees from the pool and collect them.
  - ☑ Test coverage
- Charge fees for Beefy and send them to the appropriate addresses — charge fees to accrued state fee amounts.
  - ☑ Test coverage
- Remove the liquidity.
  - ☑ Test coverage
- Readjust the ticks.
  - ☑ Test coverage
- Add the liquidity.
  - ☑ Test coverage
- Reset the state fees to zero.
  - ☑ Test coverage

Negative behavior

- Should not be callable unless the period is calm. Currently not implemented.
  - □ Negative test

Function: `panic(uint256 _minAmount0, uint256 _minAmount1)`

Allows manager to “panic”, essentially pausing deposits and removing liquidity and allowances.

Inputs

- `_minAmount0`
    - **Control**: The minimum amount of token0 in the strategy after panic.
    - **Constraints**: Checked that it is larger than balances.
    - **Impact**: If the balance is less than this, the function will revert.
- `_minAmount1`
    - **Control**: The minimum amount of token1 in the strategy after panic.
    - **Constraints**: Checked that it is larger than balances.
    - **Impact**: If the balance is less than this, the function will revert.

Branches and code coverage

Intended branches

- Remove liquidity.
    - ☑ Test coverage
- Remove allowances.
    - ☑ Test coverage
- Pause the contract.
    - ☑ Test coverage
- Check if the balances are larger than the minimum amounts.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the manager to call this function.
    - ☑ Negative test

Function: `setDeviation(int56 _deviation)`

Allows owner to set the deviation from the TWAP upon adding liquidity.

Inputs

- `_deviation`
    - **Control**: Fully controlled by the owner.
    - **Constraints**: No constraints.
    - **Impact**: The deviation from the TWAP we will allow on adding liquidity.

Branches and code coverage

Intended branches

- Update the value of `deviation`.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test

Function: `setLpToken0ToNativePath(bytes _path)`

Allows setting the path to swap the first token to the native token for fee harvesting.

Inputs

- `_path`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: No constraints.
  - **Impact**: The path to swap the first token to the native token.

Branches and code coverage

Intended branches

- Assumes the path is legitimate and sets the path to `lpToken1ToNativePath`.
  ☑ Test coverage
- Checks that the first token in the path is the same as `lpToken1`.
  ☑ Test coverage
- Checks that the last token in the path is the same as `native`.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  ☑ Negative test

Function: `setLpToken1ToNativePath(bytes _path)`

Allows setting the path to swap the second token to the native token for fee harvesting.

Inputs

- `_path`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: No constraints.

  - **Impact**: The path to swap the second token to the native token.

Branches and code coverage

Intended branches

- Assumes the path is legitimate and sets the path to `lpToken1ToNativePath`.
  - ☑ Test coverage
- Checks that the first token in the path is the same as `lpToken1`.
  - ☑ Test coverage
- Checks that the last token in the path is the same as `native`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  - ☑ Negative test

Function: `setPositionWidth(int24 _width)`

Allows owner to set the position width and readjust the positions.

Inputs

- `_width`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: No constraints.
  - **Impact**: The width of the position will be set to this value.

Branches and code coverage

Intended branches

- Remove the current liquidity.
  - ☑ Test coverage
- Set the `positionWidth` to the new value.
  - ☑ Test coverage
- Update the ticks.
  - ☑ Test coverage
- Add the previous liquidity back to the pool.
  - ☑ Test coverage

Negative behavior

- Should not be callable in periods that are not calm, so that `_removeLiquidity` and `_addLiquidity` are not called or slippaged badly.
  - □ Negative test
- Should not be callable by anyone other than the owner.
  - ☑ Negative test

Function: `uniswapV3MintCallback(uint256 amount0, uint256 amount1, bytes)`

Defines the callback function for the Uniswap V3 pool to call when minting liquidity.

Inputs

- `amount0`
  - **Control**: Controlled by the Uniswap V3 pool.
  - **Constraints**: Assumed to be a valid amount (i.e., `address(this)` can pay it).
  - **Impact**: The amount of token0 owed to the pool.
- `amount1`
  - **Control**: Controlled by the Uniswap V3 pool.
  - **Constraints**: Assumed to be a valid amount (i.e., `address(this)` can pay it).
  - **Impact**: The amount of token1 owed to the pool.

Branches and code coverage

Intended branches

- Send the owed tokens to the pool.
  - ☑ Test coverage
- Reset the `minting` state.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the pool.
  - ☑ Negative test
- Should not be callable if `minting` is false.
  - ☑ Negative test

Function: `unpause()`

Allows manager to unpause the contract.

Branches and code coverage

Intended branches

- Give the allowances back to the UniRouter.
  - ☑ Test coverage
- Unpause the contract.
  - ☑ Test coverage
- Set the ticks.
  - ☑ Test coverage
- Add the liquidity back to the pool.
  - ☑ Test coverage

Negative behavior

- Should not be called by anyone other than the manager.
  - ☑ Negative test

Function: `withdraw(uint256 _amount0, uint256 _amount1)`

Allows the withdrawal of the specified amount of tokens from the strategy as calculated by the vault.

Inputs

- `_amount0`
  - **Control**: The amount of token0 to withdraw.
  - **Constraints**: None — assumed to be within the vault's control.
  - **Impact**: The amount of token0 to withdraw.
- `_amount1`
  - **Control**: The amount of token1 to withdraw.
  - **Constraints**: None — assumed to be within the vault's control.
  - **Impact**: The amount of token1 to withdraw.

Branches and code coverage

Intended branches

- Transfer the tokens to the vault.
  - ☑ Test coverage
- Add liquidity back to the positions.
  - ☑ Test coverage

Negative behavior

- Should not allow calling under periods that are not calm. Currently not enforced.

- □ Negative test
- Should not allow anyone other than the vault to call this function.
  - ☑ Negative test
- Should not allow calling this if the strategy is paused.
  - ☑ Negative test

Function: `_addLiquidity()`

Allows the addition of liquidity to the main and alternative positions.

Branches and code coverage

Intended branches

- Check if the amounts are okay to add liquidity for the alternative position.
  - □ Test coverage
- Calculate the amounts of liquidity to add, based on the token balances and the current price.
  - ☑ Test coverage
- Check if the amounts are okay to add liquidity for the main position.
  - ☑ Test coverage
- Flip minting to true and call the pool to mint the liquidity for the main position.
  - ☑ Test coverage
- Flip minting to true and call the pool to mint the liquidity for the alternative position.
  - ☑ Test coverage

Negative behavior

- Should not be callable in periods that are not calm. Not always the case.
  - □ Negative test

Function: `_removeLiquidity()`

Function to remove liquidity from the existing positions.

Branches and code coverage

Intended branches

- Burn the existing alt-position liquidity.
  - ☑ Test coverage
- Burn the existing main-position liquidity.
  - ☑ Test coverage
- Collect the tokens from the alt position.

- ☑ Test coverage
- Collect the tokens from the main position.
    - ☑ Test coverage

Negative behavior

- Assumes it is not called in periods that are not calm.
    - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Beefy UniswapV3 contracts, we discovered 11 findings. One critical issue was found. Two were of high impact, two were of medium impact, five were of low impact, and the remaining finding was informational in nature. Beefy acknowledged all findings and implemented fixes.

6.1. Disclaimer

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 code added to the project after the version reviewed during 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 samples 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.