Zellic

Steer

Smart Contract Security Assessment

May 2, 2023

Prepared for:

Derek Barrera

Steer

Prepared by:

Aaron Esau and Junyi Wang

Zellic Inc.

Contents

About Zellic

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

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

1 Executive Summary

Zellic conducted a security assessment for Steer from March 15th to March 28th, 2023. During this engagement, Zellic reviewed Steer's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

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

- Can a vault manager lock user funds using a specially crafted `newPositions` in any way?
- What centralization risks exist for depositors?
- Can a nondepositor steal user funds from a vault?

1.2 Non-goals and Limitations

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

- 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.

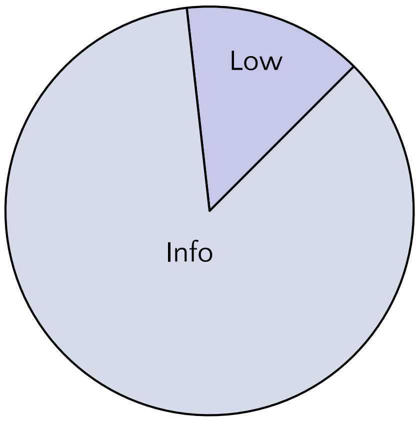
1.3 Results

During our assessment on the scoped Steer contracts, we discovered seven findings. These findings include one low severity finding and six informational findings. No critical issues were found.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	6



2 Introduction

2.1 About Steer

Steer is a decentralized compute protocol that provides a scaling solution for data processing through its robust off-chain infrastructure.

With Steer you can write cross-chain apps with 20+ programming languages, connect them to a secured data source (dynamic or static), and execute on any blockchain.

Steer infrastructure offers a multitude of possibilities like automated liquidity management, loan payments, asset management, a data availability marketplace, automated governance, oracles, and more.

To showcase the power of Steer Protocol, the Steer team has built out multichain automated liquidity management apps on the market for the next generation of automated market makers like Trident, Uniswap V3, and so forth.

2.2 Methodology

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

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

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

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

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

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

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

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

2.3 Scope

The engagement involved a review of the following targets:

Steer Contracts

Repository	https://github.com/SteerProtocol/contracts-secret
Version	contracts-secret: `98843b7b7a09c7151daa5c97abdedec08507bb46`
Programs	• SteerPeriphery • VaultRegistry • SushiBaseLiquidityManager • SushiMultiPositionLiquidityManager • SushiSinglePositionLiquidityManager

	• (Uniswap) BaseLiquidityManager • (Uniswap) MultiPositionLiquidityManager • (Uniswap) SinglePositionLiquidityManager
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau, Security Engineer
aaron@zellic.io

Junyi Wang, Security Engineer
junyi@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

March 15, 2023	Kick-off call
March 15, 2023	Start of primary review period
March 28, 2023	End of primary review period

3 Detailed Findings

3.1 The _calcSharesAndAmounts rounds amounts used down

- **Target**: BaseLiquidityManager
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `_calcSharesAndAmounts` function calculates how much the user should pay and how many shares should be minted for them. In the two branches handling the case of zero tokens of one type, the amount of tokens charged is rounded down when it should be rounded up.

```
function _calcSharesAndAmounts(
    uint256 amount0Desired,
    uint256 amount1Desired
)
    internal
    view
    returns (uint256 shares, uint256 amount0Used, uint256 amount1Used)
{
    // [ ... ]
    } else if (total0 == 0) {
        shares = FullMath.mulDiv(amount1Desired, _totalSupply, total1);
        amount1Used = FullMath.mulDiv(shares, total1, _totalSupply);
    } else if (total1 == 0) {
        shares = FullMath.mulDiv(amount0Desired, _totalSupply, total0);
        amount0Used = FullMath.mulDiv(shares, total0, _totalSupply);
    }
    // [ ... ]
```

Note that there is equivalent code in `SushiBaseLiquidityManager.deposit`.

Impact

The depositing user gains more shares than they should by a rounding error. There is a miniscule chance (if the values all match up) that when the user redeems their shares, they will gain one more unit of the token than they deposited. Note that the value of one unit of token is insignificant, however, since tokens usually have a large denominator.

Recommendations

Round up the `amount1Used` and `amount0Used` in the above branches.

Remediation

This issue has been acknowledged by Steer, and a fix was implemented in commit 2986c269.

3.2 The `strategyCreator` is not verified in `createVaultAndStrategy`

- **Target**: SteerPeriphery
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

In the `createVaultAndStrategy`, the `strategyCreator` parameter is not checked and is passed into `strategyRegistry.createStrategy`.

```
function createVaultAndStrategy(
    address strategyCreator,
    string memory name,
    string memory execBundle,
    uint128 maxGasCost,
    uint128 maxGasPerAction,
    bytes memory params,
    string memory beaconName,
    address vaultManager,
    string memory payloadIpfs
) external payable returns (uint256 tokenId, address newVault) {
    tokenId = IStrategyRegistry(strategyRegistry).createStrategy(
        strategyCreator,
        name,
        execBundle,
        maxGasCost,
        maxGasPerAction
    );
    // [...]
}
```

The `strategyCreator` is not subsequently checked in `strategyRegistry.createStrategy`.

Impact

It is possible to create a strategy using someone else's address, which is not necessarily a major concern. Falsifying the strategy creator will lead to the following:

1. The true creator of the strategy losing control of the strategy, since the NFT is minted to the false creator; and
2. The true creator being unable to collect fees. The vault is empty until someone deposits into the vault.

No funds are at risk from a strategy with a false creator. However, it does have the potential for confusion.

Recommendations

Consider enforcing the strategy creator to always be `msg.sender`. However, this is not strictly necessary.

Remediation

Steer provided the following response:

> Issue 3.2 relates to the verification of the strategyCreator in the createVaultAndStrategy function. Our team's goal is to provide creators with a flexible approach that allows them to generate a strategy from any address and specify the owner of that strategy, whether it is the same address or a different one.
>
> To optimize the user experience, our dapp requires that both the vault and strategy are created in a single transaction using periphery's createVaultAndStrategy. To achieve this, we pass the creator as a parameter instead of using msg.sender.

3.3 Ability to drain SteerPeriphery of tokens

- **Target**: SteerPeriphery
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

This is the `_deposit` function of SteerPeriphery:

```
function _deposit(
    address vaultAddress-,
    uint256 amount0Desired,
    uint256 amount1Desired,
    uint256 amount0Min,
    uint256 amount1Min,
    address to
) internal returns (uint256) {
    IMultiPositionManager vaultInstance = IMultiPositionManager(
        vaultAddress
    );
    IERC20 token0 = IERC20(vaultInstance.token0());
    IERC20 token1 = IERC20(vaultInstance.token1());
    if (amount0Desired > 0)
        token0.safeTransferFrom(msg.sender, address(this),
    amount0Desired);
    if (amount1Desired > 0)
        token1.safeTransferFrom(msg.sender, address(this),
    amount1Desired);
    token0.approve(vaultAddress, amount0Desired);
    token1.approve(vaultAddress, amount1Desired);

    (uint256 share, uint256 amount0, uint256 amount1) = vaultInstance
        .deposit(
            amount0Desired,
            amount1Desired,
            amount0Min,
            amount1Min,
            to
        );
    if (amount0Desired > amount0) {
        token0.approve(vaultAddress, 0);
        token0.safeTransfer(msg.sender, amount0Desired - amount0);
    }
    if (amount1Desired > amount1) {
        token1.approve(vaultAddress, 0);
        token1.safeTransfer(msg.sender, amount1Desired - amount1);
    }

    return share;
}
```

Because the `vaultAddress` parameter is user input, there are a few vectors an attacker could use to drain the contract of the two tokens — the simplest being passing a malicious vault contract that returns 0 for `amount0` and `amount1` such that the refund transfers double the `amount0Desired` and `amount1Desired` sent to the contract.

Impact

Should the contract hold any tokens between transactions, an attacker could potentially drain the contract of tokens.

Fortunately, this finding has no impact because SteerPeriphery only transiently holds tokens (only within a single transaction).

Recommendations

Prominently document that SteerPeriphery should never hold tokens.

Remediation

This issue has been acknowledged by Steer, and a fix was implemented in commit 0e3ed983.

3.4 No validation on `tokenId` in `createStrategy`

- **Target**: VaultRegistry
- **Category**: Business Logic
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

In the `createVault` function, the `_tokenId` parameter is passed to `_addLinkedVaultsEnumeration` without validation:

```
function createVault(
    bytes memory _params,
    uint256 _tokenId,
    string memory _beaconName,
    address _vaultManager,
    string memory _payloadIpfs
) external whenNotPaused returns (address) {
    // [ ... ]
    _addLinkedVaultsEnumeration(
        _tokenId,
        address(newVault),
        _payloadIpfs,
        _beaconName
    );
}
```

In the `_addLinkedVaultsEnumeration` function, the `_tokenId` is stored without validation in the `VaultData` struct.

```
function _addLinkedVaultsEnumeration(
    uint256 _tokenId,
    address _deployedAddress,
    string memory _payloadIpfs,
    string memory _beaconName
) internal {
    // Get the current count of how many vaults have been created from
    this strategy
    uint256 currentCount = linkedVaultCounts[_tokenId];
    // Using _tokenId and count as map keys, add the vault to the list of
    linked vaults
    linkedVaults[_tokenId][currentCount] = _deployedAddress;
    // Increment the count of how many vaults have been created from a
    given strategy
    linkedVaultCounts[_tokenId] = currentCount + 1;
    // Store any vault specific data via the _deployedAddress
    vaults[_deployedAddress] = VaultData({
        state: VaultState.PendingThreshold,
        tokenId: _tokenId, // no validation on _tokenId
        // [ ... ]
    });
}
```

Impact

The `VaultData` struct that contains the `_tokenId` is used in `getAvailableForTransaction` to fetch the `RegisteredStrategy` struct via `strategyRegistry.getRegisteredStrategy(vaultInfo.tokenId)`.

Given a vault with a nonexistent `tokenId`, this would return a null `RegisteredStrategy` struct, which would then cause a reversion from `require(tx.gasprice ≤ info.maxGasCost, "Gas too expensive.");`.

Recommendations

Assert that the strategy token ID is registered.

Remediation

This issue has been acknowledged by Steer, and a fix was implemented in commit b483149a.

3.5 Deposits do not check vault type

- **Target**: SteerPeriphery
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

The `deposit` function does not check that the `vaultAddress` parameter is a valid vault.

```
function deposit(
    address vaultAddress,
    uint256 amount0Desired,
    uint256 amount1Desired,
    uint256 amount0Min,
    uint256 amount1Min,
    address to
) external {
    _deposit(
        vaultAddress, // not checked for validity
        amount0Desired,
        amount1Desired,
        amount0Min,
        amount1Min,
        to
    );
}
```

Impact

A deposit could be made to an invalid vault type.

Recommendations

Use `IVaultRegistry(vaultRegistry).beaconTypes(vault)` to ensure the vault was registered properly (i.e., has an associated beacon type).

Consider also checking the vault state to ensure that it is approved or pending threshold.

Remediation

This issue has been acknowledged by Steer, and a fix was implemented in commit 077d6836.

3.6 Missing `twapInterval` and `maxTickChange` validation

- **Target**: SushiBaseLiquidityManager, BaseLiquidityManager
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

The `initialize` functions of the Sushi and Uniswap vault base contracts do not include any assertions to ensure that the `twapInterval` and `maxTickChange` parameters are within a reasonable range.

Impact

A user could unintentionally deploy a pool with parameters that leave it vulnerable to oracle manipulation attacks.

Recommendations

Assert that the `twapInterval` and `maxTickChange` parameters are within reasonable ranges.

Remediation

This issue has been acknowledged by Steer, and a fix was implemented in commit f2881e83 for the `SushiBaseLiquidityManager`.

A fix was implemented in commits ac11ba56 and f7ceb734 for the `BaseLiquidityManager`.

3.7 Missing zero check on `vaultAddress` in `depositAndStake`

- **Target**: SteerPeriphery
- **Category**: Coding Mistakes
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

The following assertion in SteerPeriphery's `depositAndStake` function can be bypassed if the `vaultAddress` parameter is 0 and the `poolId` is not registered in StakingRewards:

```
require(
    IStakingRewards(stakingRewards).getPool(poolId).stakingToken ==
        vaultAddress,
    "Incorrect pool id"
);
```

Impact

In StakingRewards' `_stake` function, the following assertion would assume the staking period is over since `pool.end` would be 0 and thereby prevent any potentially malicious behavior:

```
require(block.timestamp < pool.end, "Staking Period is over");
```

Recommendations

Assert that the `vaultAddress` is nonzero in `depositAndStake`:

```
require(
    vaultAddress ≠ 0 &&
    IStakingRewards(stakingRewards).getPool(poolId).stakingToken ==
        vaultAddress,
    "Incorrect pool id"
);
```

Remediation

This issue has been acknowledged by Steer, and a fix was implemented in commit e97dbc57.

4 Discussion

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

4.1 Contracts only evaluated for ERC-20

It is important that tokens used in the audited contracts strictly adhere to the ERC-20 standard; although the protocol may function properly with tokens following other standards, some standards introduce functionality that causes the protocol to be vulnerable.

For example, although the ERC-777 standard may simply be extending the ERC-20 standard, it allows attackers to execute arbitrary code on transfer or mint, which could potentially enable reentrancy attacks.

This audit assumes that only the ERC-20 standard will be used, as all contracts use the IERC20 interface.

4.2 SteerPeriphery can be upgraded

Note that the SteerPeriphery contract can be upgraded and should be used with caution; the owner could potentially front-run a call to `deposit` to take advantage of the token approval to steal funds.

We recommend ensuring the owner is a governance contract.

4.3 Vault manager needs to be trusted

Note that the vault manager should be trusted — whether it is a smart contract or a wallet — since it is capable of making potentially risky or negative transactions or MEVing user funds.

5 Threat Model

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

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

5.1 Module: BaseLiquidityManager.sol

Function: `deposit(uint256 amount0Desired, uint256 amount1Desired, uint256 amount0Min, uint256 amount1Min, address to)`

Deposits tokens into the vault and mints shares for the user.

Assumptions

- The utility functions correctly determine
  - The total number of shares.
  - The total number of `token0`, `token1`.

Assertions

The arguments `amount0Min`, `amount1Min` have no security implications.

- They are used for slippage protection. The user may fill whatever they like; it will only lead to the transaction being reverted if they fill an out-of-range value.

The argument `to` has no security implications.

- Since the user is paying for the shares, the user may credit them to whoever they like.

The amount of shares minted to a user does not exceed the share of tokens deposited.

- `deposited_total0 ≥ minted_shares / (old_totalSupply + minted_shares) * (old_total0 + deposited_total0)` holds after the transaction.

- `deposited_total1 ≥ minted_shares / (old_totalSupply + minted_shares) * (old_total1 + deposited_total1)` holds after the transaction.

Below is the `_calcSharesAndAmounts` function, which has four branches. We will prove that the above holds in all four branches.

```
function _calcSharesAndAmounts(
    uint256 amount0Desired,
    uint256 amount1Desired
)
    internal
    view
    returns (uint256 shares, uint256 amount0Used, uint256 amount1Used)
{
    uint256 _totalSupply = totalSupply();
    (uint256 total0, uint256 total1) = getTotalAmounts();
    assert(_totalSupply == 0 || total0 > 0 || total1 > 0);
    if (_totalSupply == 0) {
        amount0Used = amount0Desired;
        amount1Used = amount1Desired;
        shares = Math.max(amount0Used, amount1Used);
    } else if (total0 == 0) {
        shares = FullMath.mulDiv(amount1Desired, _totalSupply, total1);
        amount1Used = FullMath.mulDiv(shares, total1, _totalSupply);
    } else if (total1 == 0) {
        shares = FullMath.mulDiv(amount0Desired, _totalSupply, total0);
        amount0Used = FullMath.mulDiv(shares, total0, _totalSupply);
    } else {
        uint256 cross = Math.min(
            amount0Desired.mul(total1),
            amount1Desired.mul(total0)
        );
        require(cross > 0, "C");
        amount0Used = ((cross - 1) / total1) + 1;
        amount1Used = ((cross - 1) / total0) + 1;
        shares = FullMath.mulDiv(cross, _totalSupply, total0) / total1;
    }
}
```

- Branch `_totalSupply == 0`:
  - `minted_shares / (old_totalSupply + minted_shares) == 1`, since `old_totalSupply == 0`.

    - `deposited_total0 ≥ deposited_total0` and `deposited_total1 ≥ deposited_total1` trivially.
    - `old_total0 == 0` and `old_total1 == 0` by assumption that token number tracking is correct.
    - It holds that `deposited_total0 ≥ 1 * (0 + deposited_total0)`.
    - Therefore `deposited_total0 ≥ minted_shares / (old_totalSupply + minted_shares) * (old_total0 + deposited_total0)`.
- Branch `total0 == 0`:
    - `deposited_total1 ~= minted_shares * old_total1 / old_totalSupply` where ~= is "approximately".
    - `deposited_total1 * old_totalSupply ~= minted_shares * old_total1`.
    - `deposited_total1 * old_totalSupply + minted_shares * deposited_total1 ~= minted_shares * old_total1 + minted_shares * deposited_total1`.
    - `deposited_total1 * (old_totalSupply + minted_shares) ~= minted_shares * (old_total1 + deposited_total1)`.
    - `deposited_total1 ~= minted_shares / (old_totalSupply + minted_shares) * (old_total1 + deposited_total1)`.
    - Note that the user actually does gain more shares than they should, by a rounding error, since `amount1Used = FullMath.mulDiv(shares, total1, _totalSupply)` rounds down.
- Branch `total1 == 0`:
    - Symmetrical to above case.
- Branch `else`:
    - `deposited_total0 ≥ cross / old_total1` by code `amount0Used = ((cross - 1) / total1) + 1`.
    - `minted_shares ≤ cross * old_totalSupply / old_total0 / old_total1` by code `shares = FullMath.mulDiv(cross, _totalSupply, total0) / total1`.
    - `minted_shares ≤ (cross / old_total1) * old_totalSupply / old_total0` by rearranging.
    - `minted_shares ≤ deposited_total0 * old_totalSupply / old_total0` by substitution of the first inequality.
    - `deposited_total0 * old_totalSupply ≥ minted_shares * old_total0` by rearranging.
    - With a similar transformation as for the second branch, we have the following:
    - `deposited_total0 ≥ minted_shares / (old_totalSupply + minted_shares) * (old_total0 + deposited_total0)`.
    - A symmetrical argument applies to `deposited_total1`.
- Therefore, the amount of shares minted is less than the amount contributed

(module rounding error).

Branches and code coverage (including function calls)

Intended branches

- ☐ Reverts if the amount used values exceed the values used for slippage protection.
- ☑ Mints the right number of shares when the vault is empty.
- ☐ Mints the right number of shares when the vault has only one token type.
- ☐ Mints the right number of shares when the vault has both token types.

Function: `emergencyBurn(int24 tickLower, int24 tickUpper, uint128 liquidity)`

Removes liquidity from the Uniswap pool in case of an emergency and collects it in the vault. This function is intended to be called by the steer role only.

Skips collection of fees and does not check for the contract being paused.

Inputs

- `tickLower`
  - **Control**: Full.
  - **Constraints**: Must be the `tickLower` of an existing position in the pool.
  - **Impact**: The lower tick of the position to remove liquidity from.
- `tickUpper`
  - **Control**: Full.
  - **Constraints**: Must be the `tickUpper` of an existing position in the pool.
  - **Impact**: The upper tick of the position to remove liquidity from.
- `liquidity`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `liquidity` of the position in the pool or it will revert.
  - **Impact**: The amount of liquidity to remove from the position and withdraw to the vault.

Branches and code coverage (including function calls)

Intended branches

- Removes liquidity from the position and collects it in the vault.
  - ☐ Test coverage

Negative behavior

- Position does not exist.
  - ☐ Negative test
- Caller does not have `STEER_ROLE` role.
  - ☐ Negative test
- `liquidity` is greater than the position's `liquidity`.
  - ☐ Negative test

Function call analysis

- `pool.burn(tickLower, tickUpper, liquidity)`
  - **What is controllable?** The `tickLower`, `tickUpper`, and `liquidity` inputs.
  - **If return value controllable, how is it used and how can it go wrong?** These are the parameters that specify the position and the amount of liquidity to remove from the position. If the parameters are invalid, it will revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The steer role will be unable to burn liquidity in the case of an emergency if this call reverts. However, the steer role can call the function again with different values and attempt to avoid a revert.
- `pool.collect( … )`
  - **What is controllable?** The `tickLower` and `tickUpper` inputs.
  - **If return value controllable, how is it used and how can it go wrong?** These are the parameters that specify the position. If the parameters are invalid, it will revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The steer role will be unable to collect the liquidity in the case of an emergency if this call reverts. However, the steer role can call the function again with different values and attempt to avoid a revert.

Function: `pause()`

Calls the OpenZepplin PausableUpgradeable contract's `_pause` function to flag the contract as paused. This function is intended to be called by the steer role only.

Branches and code coverage (including function calls)

Intended branches

- Flags the contract as paused.
  - ☐ Test coverage

Negative behavior

- Contract is already paused (this is enforced by `_pause`'s `whenNotPaused` modifier).
  - ☐ Negative test
- Caller does not have `STEER_ROLE` role.
  - ☐ Negative test

Function call analysis

- `_pause()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The contract will not be flagged as paused. This will not happen unless the negative behavior is intended, per the PausableUpgradeable contract.

Function: `setMaxTotalSupply(uint256 _maxTotalSupply)`

Sets the maximum total supply of the vault, which is used to limit the amount of tokens that can be deposited. This function is intended to be called by the governance role only.

Inputs

- `_maxTotalSupply`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The maximum total supply of the vault. The governance caller could prevent users from depositing by setting the `maxTotalSupply` to a value lower than or equal to the current supply.

Branches and code coverage (including function calls)

Intended branches

- Changes the maximum supply setting.
  - ☐ Test coverage

Negative behavior

- Caller does not have `GOVERNANCE_ROLE` role.
  - ☐ Negative test
- Contract is paused.
  - ☐ Negative test

Function: `steerCollectFees(uint256 amount0, uint256 amount1, address to )`

Collects fees from the Uniswap pool and sends them to the `to` address.

Inputs

- `amount0`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees0` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the first token.
- `amount1`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees1` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the second token.
- `to`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The address to which the fees are sent.

Branches and code coverage (including function calls)

Intended branches

- Caller collects fees for both tokens to the `to` address.
  - [x] Test coverage

Negative behavior

- Caller does not have `STEER_ROLE` role.
  - [ ] Negative test
- Contract is paused.
  - [ ] Negative test
- `amount0` is greater than `accruedFees0`.
  - [ ] Negative test
- `amount1` is greater than `accruedFees1`.
  - [ ] Negative test

Function: `strategistCollectFees(uint256 amount0, uint256 amount1, address to)`

Collects fees from the Uniswap pool dedicated to the strategist and sends them to the `to` address.

Inputs

- `amount0`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees0` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the first token.
- `amount1`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees1` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the second token.
- `to`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The address to which the fees are sent.

Branches and code coverage (including function calls)

Intended branches

- Caller collects fees for both tokens to the `to` address.
  - ☑ Test coverage

Negative behavior

- Caller is not the strategist.
  - ☑ Negative test
- Contract is paused.
  - ☐ Negative test
- `amount0` is greater than `accruedFees0`.
  - ☐ Negative test
- `amount1` is greater than `accruedFees1`.
  - ☐ Negative test

Function call analysis

- `IBareVaultRegistry(vaultRegistry).getStrategyCreatorForVault(address(this))`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Strategist would be unable to collect fees if it reverted. Reentering is impossible given the `vaultRegistry` is a VaultRegistry contract, but it would have no impact in this context regardless.

Function: `uniswapV3MintCallback(uint256 amount0, uint256 amount1, byte[] None)`

This function is called by the Uniswap V3 pool when liquidity is minted. It is used to pay UniswapV3Pool back tokens it is owed.

Inputs

- `amount0`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `token0` balance of the vault.
  - **Impact**: The amount of `token0` to pay back to Uniswap V3.
- `amount1`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `token1` balance of the vault.
  - **Impact**: The amount of `token1` to pay back to Uniswap V3.

Branches and code coverage (including function calls)

Intended branches

- Transfers tokens to the Uniswap V3 pool. Indirectly tested when minting liquidity.
  - [x] Test coverage

Negative behavior

- `msg.sender` is not the pool.
  - [ ] Negative test

Function call analysis

- `_transferTokens(msg.sender, amount0, amount1)`

  - **What is controllable?** `amount0`, `amount1`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `uniswapV3SwapCallback(uint256 amount0, uint256 amount1, byte[] None)`

This function is called by the Uniswap V3 pool when performing a swap. It is used to pay UniswapV3Pool back tokens it is owed.

Inputs

- `amount0`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `token0` balance of the vault.
  - **Impact**: The amount of `token0` to pay back to Uniswap V3.
- `amount1`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `token1` balance of the vault.
  - **Impact**: The amount of `token1` to pay back to Uniswap V3.

Branches and code coverage (including function calls)

Intended branches

- Transfers tokens to the Uniswap V3 pool. Indirectly tested when performing a swap.
  - ☑ Test coverage

Negative behavior

- `msg.sender` is not the pool.
  - ☐ Negative test

Function call analysis

- `token0.safeTransfer(msg.sender, uint256(amount0Wanted))`
  - **What is controllable?** `amount0Wanted`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `token1.safeTransfer(msg.sender, uint256(amount1Wanted))`

    - **What is controllable?** `amount1Wanted`.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `unpause()`

Calls the OpenZepplin PausableUpgradeable contract's `_unpause` function to flag the contract as unpaused. This function is intended to be called by the steer role only.

Branches and code coverage (including function calls)

Intended branches

- Flags the contract as unpaused.
  - ☐ Test coverage

Negative behavior

- Contract is already unpaused (this is enforced by `_unpause`'s `whenPaused` modifier).
  - ☐ Negative test
- Caller does not have `STEER_ROLE` role.
  - ☐ Negative test

Function call analysis

- `_unpause()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The contract will not be flagged as unpaused. This will not happen unless the negative behavior is intended, per the PausableUpgradeable contract.

Function: `withdraw(uint256 shares, uint256 amount0Min, uint256 amount1Min, address to)`

Withdraws the proportion of tokens in the vault and in the pool that corresponds to the shares burned.

Inputs

- `shares`
  - **Control**: Full.

    - **Constraints**: Must be less than or equal to the number of shares owned by the sender. Otherwise, `_burn` will revert.
    - **Impact**: The amount of tokens withdrawn will be proportional to the number of `shares` burned.
- `amount0Min`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: If the amount of `token0` withdrawn is less than `amount0Min`, the function will revert.
- `amount1Min`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: If the amount of `token1` withdrawn is less than `amount1Min`, the function will revert.
- `to`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The withdrawn tokens will be sent to `to`.

Branches and code coverage (including function calls)

Intended branches

- Withdraws tokens and burns shares.
  ☑ Test coverage

Negative behavior

- Reverts if `shares` is zero.
  ☐ Negative test
- Reverts if `shares` is greater than the number of shares owned by the sender.
  ☐ Negative test
- Reverts if the amount of `token0` withdrawn is less than `amount0Min`.
  ☐ Negative test
- Reverts if the amount of `token1` withdrawn is less than `amount1Min`.
  ☐ Negative test

Function call analysis

- `totalSupply()`
    - **What is controllable?** None.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.

  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions.
- `_burn(msg.sender, shares)`
  - **What is controllable?** `msg.sender` and `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions.
- `_burnAndCollect(shares, _totalSupply)`
  - **What is controllable?** `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions.
- `token0.safeTransfer(to, amount0)`
  - **What is controllable?** `to`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions because ERC-20 tokens are being used.
- `token1.safeTransfer(to, amount1)`
  - **What is controllable?** `to`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions because ERC-20 tokens are being used.

5.2 Module: MultiPositionLiquidityManager.sol

Function: `poke()`

Updates fees owed to each vault position. Given too many vault positions, this function may run out of gas.

Branches and code coverage (including function calls)

Intended branches

- Updates fees owed to each vault position.

☑ Test coverage

Function call analysis

- `pool.burn(_lowerTick, _upperTick, 0)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Fees will not be updated. Function cannot reenter because `pool` is the trident liquidity pool contract.

Function: `tend(uint256 totalWeight, LiquidityPositions newPositions, byte[] timeSensitiveData)`

Adjusts the vault's positions to match the new positions.

Inputs

- `totalWeight`
  - **Control**: Full.
  - **Constraints**: Must be low enough that multiplication does not result in overflow.
  - **Impact**: The share of liquidity we want deposited, multiplied by 10,000.
- `newPositions`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The new positions to be set.
- `timeSensitiveData`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Encoded information of the `swapAmount` and `sqrtPriceLimitX96` used to mitigate MEV attacks.

Branches and code coverage (including function calls)

Intended branches

- Updates the position.
  ☑ Test coverage
- Performs a swap.
  ☑ Test coverage
- Creates a new position.

- [x] Test coverage

Negative behavior

- Reverts if `msg.sender` does not have the `MANAGER_ROLE` role.
  - [ ] Negative test
- Reverts if the contract is paused.
  - [ ] Negative test
- Reverts if TWAP-derived tick is too far from the current tick.
  - [x] Negative test

Function call analysis

- `pool.slot0()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; compared against TWAP.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_checkVolatility(currentTick)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_burnAndCollect(1, 1)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_updatePositions(newPositions)`
  - **What is controllable?** `newPositions`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_swap( ... )`
  - **What is controllable?** `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `pool.slot0()`
  - **What is controllable?** None.

  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_setBins( ... )`
  - **What is controllable?** `FullMath.mulDiv(balance0, totalWeight, 1e4)`, `FullMath.mulDiv(balance1, totalWeight, 1e4)`, and `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.

5.3 Module: SinglePositionLiquidityManager.sol

Function: `poke()`

Updates fees owed to each vault position.

Branches and code coverage (including function calls)

Intended branches

- Updates fees owed to each vault position.
  ☑ Test coverage

Function call analysis

- `pool.burn(_lowerTick, _upperTick, 0)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Fees will not be updated. Function cannot reenter because `pool` is the trident liquidity pool contract.

Function: `tend(uint256 totalWeight, LiquidityPositions newPositions, byte[] timeSensitiveData)`

Adjusts the vault's positions to match the new positions.

Inputs

- `totalWeight`
  - **Control**: Full.
  - **Constraints**: Must be low enough that multiplication does not result in

overflow.
    - **Impact**: The share of liquidity we want deposited, multiplied by 10,000.
- `newPositions`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The new positions to be set.
- `timeSensitiveData`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: Encoded information of the `swapAmount` and `sqrtPriceLimitX96` used to mitigate MEV attacks.

Branches and code coverage (including function calls)

Intended branches

- Updates the position.
    - [x] Test coverage
- Performs a swap.
    - [x] Test coverage
- Creates a new position.
    - [x] Test coverage

Negative behavior

- Reverts if `msg.sender` does not have the `MANAGER_ROLE` role.
    - [ ] Negative test
- Reverts if the contract is paused.
    - [ ] Negative test
- Reverts if TWAP-derived tick is too far from the current tick.
    - [x] Negative test

Function call analysis

- `pool.slot0()`
    - **What is controllable?** None.
    - **If return value controllable, how is it used and how can it go wrong?** N/A; compared against TWAP.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_checkVolatility(currentTick)`
    - **What is controllable?** None.

  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_burnAndCollect(1, 1)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_updatePositions(newPositions)`
  - **What is controllable?** `newPositions`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_swap( ... )`
  - **What is controllable?** `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `pool.slot0()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_setBins( ... )`
  - **What is controllable?** `FullMath.mulDiv(balance0, totalWeight, 1e4)`, `FullMath.mulDiv(balance1, totalWeight, 1e4)`, and `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.

5.4 Module: SteerPeriphery.sol

Function: `createVaultAndDepositGas(uint256 tokenId, byte[] params, string beaconName, address vaultManager, string payloadIpfs)`

Creates a new vault and deposits gas into it. These two actions can be done separately, but this function is provided for convenience (a helper function).

Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The vault will be created with the `tokenId` setting.
- `params`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The vault will be created passing in the `params` setting.
- `beaconName`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function (e.g., the beacon must exist).
  - **Impact**: The beacon used to create the vault will be the one with the name `beaconName`.
- `vaultManager`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The manager of the vault.
- `payloadIpfs`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The `payloadIpfs` setting to be stored in the `VaultData`.

Branches and code coverage (including function calls)

Intended branches

- Creates a vault and deposits gas.
  - ☑ Test coverage

Negative behavior

- Invalid beacon name (indirectly tested in `createVault`).
  - ☑ Negative test

Function call analysis

- `IVaultRegistry(vaultRegistry).createVault(params, tokenId, beaconName, va ultManager, payloadIpfs)`
  - **What is controllable?** `tokenId`, `params`, `beaconName`, `vaultManager`, and `payl`

    `oadIpfs`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The vault will not be created if it reverts. If it reenters, it is equivalent to calling `createVaultAndDepositGas`.

Function: `createVaultAndStrategy(address strategyCreator, string name, string execBundle, uint128 maxGasCost, uint128 maxGasPerAction, byte[] params, string beaconName, address vaultManager, string payloadIpfs)`

Creates a new vault and strategy and registers them in the registry. This function is intended to be a helper function, and all calls it makes may be made independently of this function.

Inputs

- `strategyCreator`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createStrategy` function.
  - **Impact**: The `strategyCreator` value passed into `createStrategy`.
- `name`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createStrategy` function.
  - **Impact**: The `name` value passed into `createStrategy`.
- `execBundle`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createStrategy` function.
  - **Impact**: The `execBundle` value passed into `createStrategy`.
- `maxGasCost`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createStrategy` function.
  - **Impact**: The `maxGasCost` value passed into `createStrategy`.
- `maxGasPerAction`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createStrategy` function.
  - **Impact**: The `maxGasPerAction` value passed into `createStrategy`.
- `params`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The `params` value passed into `createVault`.

- `beaconName`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The `beaconName` value passed into `createVault`.
- `vaultManager`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The `vaultManager` value passed into `createVault`.
- `payloadIpfs`
  - **Control**: Full.
  - **Constraints**: Only those imposed by the `createVault` function.
  - **Impact**: The `payloadIpfs` value passed into `createVault`.

Branches and code coverage (including function calls)

Intended branches

- Creates a vault and strategy.
  - ☑ Test coverage

Negative behavior

- Reverts if `createStrategy` reverts.
  - ☐ Negative test
- Reverts if `createVault` reverts.
  - ☐ Negative test

Function call analysis

- `IStrategyRegistry(strategyRegistry).createStrategy(strategyCreator, name, execBundle, maxGasCost, maxGasPerAction)`
  - **What is controllable?** `strategyCreator`, `name`, `execBundle`, `maxGasCost`, and `maxGasPerAction`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The function reverts. Reentrancy has no impact because all function calls can be made independently of this function.
- `IVaultRegistry(vaultRegistry).createVault(params, tokenId, beaconName, vaultManager, payloadIpfs)`
  - **What is controllable?** `params`, `tokenId`, `beaconName`, `vaultManager`, and `payloadIpfs`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.

  - **What happens if it reverts, reenters, or does other unusual control flow?** The function reverts. Reentrancy has no impact because all function calls can be made independently of this function.

Function: `_deposit(address vaultAddress, uint256 amount0Desired, uint256 amount1Desired, uint256 amount0Min, uint256 amount1Min, address to)`

Deposit tokens into a vault, minting shares.

Assumptions

- `SteerPeriphery` holds no tokens except transiently in the middle of transactions.
- Steer-provided vaults have correct `deposit` functions.
- Steer-provided vaults do not transfer tokens back to `SteerPeriphery`.

Assertions

Tokens cannot be stolen.

- Either
  - `vaultAddress` is a real vault.
  - `vaultAddress` is an attacker-supplied contract.
- When `vaultAddress` is a real vault,
  - Token contracts used are real ERC-20 contracts provided by the vault.
  - If not enough tokens are supplied, the vault's `deposit` will revert when attempting to transfer tokens to itself.
  - If too many tokens are returned to the user, this function will revert due to insufficient tokens to transfer.
- When `vaultAddress` is an attacker-supplied contract,
  - In this function, the only addresses involved are `SteerPeriphery`, `vaultAddress`, and the attacker's address.
  - As `SteerPeriphery` holds no tokens, no tokens can be stolen.
- This holds regardless of the values of the other parameters.

Branches and code coverage (including function calls)

Intended branches

- ☑ Deposits tokens and mints shares correctly, given a real vault address.
- ☐ Reverts if the minimum amount of given tokens is not met.

Negative behavior

☐ Reverts if the user does not have enough tokens.

5.5 Module: SushiBaseLiquidityManager.sol

Function: `deposit(uint256 amount0Desired, uint256 amount1Desired, uint256 amount0Min, uint256 amount1Min, address to)`

Deposits tokens into the vault and mints shares for the user.

Assumptions

- The utility functions correctly determine
  - The total number of shares.
  - The total number of `token0`, `token1`.

Assertions

See `BaseLiquidityManager.deposit`. This is semantically equivalent.

Branches and code coverage (including function calls)

Intended branches

☐ Reverts if the amount used values exceed the values used for slippage protection.

☑ Mints the right number of shares when the vault is empty.

☐ Mints the right number of shares when the vault has only one token type.

☐ Mints the right number of shares when the vault has both token types.

Function: `emergencyBurn(int24 tickLower, int24 tickUpper, uint128 liquidity)`

Removes liquidity from the trident pool in case of an emergency and collects it in the vault. This function is intended to be called by the steer role only.

Skips collection of fees and does not check for the contract being paused.

Inputs

- `tickLower`
  - **Control**: Full.
  - **Constraints**: Must be the `tickLower` of an existing position in the pool.
  - **Impact**: The lower tick of the position to remove liquidity from.

- `tickUpper`
  - **Control**: Full.
  - **Constraints**: Must be the `tickUpper` of an existing position in the pool.
  - **Impact**: The upper tick of the position to remove liquidity from.
- `liquidity`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `liquidity` of the position in the pool, or it will revert in `pool._updatePosition` (called by `pool.burn`) when subtracting `amount` from `position.liquidity`.
  - **Impact**: The amount of liquidity to remove from the position and withdraw to the vault.

Branches and code coverage (including function calls)

Intended branches

- Removes liquidity from the position and collects it in the vault.
  - ☐ Test coverage

Negative behavior

- Position does not exist.
  - ☐ Negative test
- Caller does not have `STEER_ROLE` role.
  - ☐ Negative test
- `liquidity` is greater than the position's `liquidity`.
  - ☐ Negative test

Function call analysis

- `pool.burn(tickLower, tickUpper, liquidity)`
  - **What is controllable?** The `tickLower`, `tickUpper`, and `liquidity` inputs.
  - **If return value controllable, how is it used and how can it go wrong?** These are the parameters that specify the position and the amount of liquidity to remove from the position. If the parameters are invalid, it will revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The steer role will be unable to burn liquidity in the case of an emergency if this call reverts. However, the steer role can call the function again with different values and attempt to avoid a revert.
- `pool.collect( … )`
  - **What is controllable?** The `tickLower` and `tickUpper` inputs.
  - **If return value controllable, how is it used and how can it go wrong?** These

are the parameters that specify the position. If the parameters are invalid, it will revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The steer role will be unable to collect the liquidity in the case of an emergency if this call reverts. However, the steer role can call the function again with different values and attempt to avoid a revert.

Function: `pause()`

Calls the OpenZepplin PausableUpgradeable contract's `_pause` function to flag the contract as paused. This function is intended to be called by the steer role only.

Branches and code coverage (including function calls)

Intended branches

- Flags the contract as paused.
  - ☐ Test coverage

Negative behavior

- Contract is already paused (this is enforced by `_pause`'s `whenNotPaused` modifier).
  - ☐ Negative test
- Caller does not have `STEER_ROLE` role.
  - ☐ Negative test

Function call analysis

- `_pause()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The contract will not be flagged as paused. This will not happen unless the negative behavior is intended, per the PausableUpgradeable contract.

Function: `setMaxTotalSupply(uint256 _maxTotalSupply)`

Sets the maximum total supply of the vault, which is used to limit the amount of tokens that can be deposited. This function is intended to be called by the governance role only.

Inputs

- `_maxTotalSupply`

- **Control**: Full.
- **Constraints**: None.
- **Impact**: The maximum total supply of the vault. The governance caller could prevent users from depositing by setting the `maxTotalSupply` to a value lower than or equal to the current supply.

Branches and code coverage (including function calls)

Intended branches

- Changes the maximum supply setting.
  - ☐ Test coverage

Negative behavior

- Caller does not have `GOVERNANCE_ROLE` role.
  - ☐ Negative test
- Contract is paused.
  - ☐ Negative test

Function: `steerCollectFees(uint256 amount0, uint256 amount1, address to)`

Collects fees from the trident pool and sends them to the `to` address.

Inputs

- `amount0`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees0` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the first token.
- `amount1`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees1` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the second token.
- `to`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The address to which the fees are sent.

Branches and code coverage (including function calls)

Intended branches

- Caller collects fees for both tokens to the `to` address.
  - [x] Test coverage

Negative behavior

- Caller does not have `STEER_ROLE` role.
  - [ ] Negative test
- Contract is paused.
  - [ ] Negative test
- `amount0` is greater than `accruedFees0`.
  - [ ] Negative test
- `amount1` is greater than `accruedFees1`.
  - [ ] Negative test

Function: `strategistCollectFees(uint256 amount0, uint256 amount1, address to)`

Collects fees from the trident pool dedicated to the strategist and sends them to the `to` address.

Inputs

- `amount0`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees0` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the first token.
- `amount1`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `accruedFees1` variable or it will revert in `_collectFees` when subtracting.
  - **Impact**: The amount of fees collected for the second token.
- `to`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The address to which the fees are sent.

Branches and code coverage (including function calls)

Intended branches

- Caller collects fees for both tokens to the `to` address.
  - ☑ Test coverage

Negative behavior

- Caller is not the strategist.
  - ☑ Negative test
- Contract is paused.
  - ☐ Negative test
- `amount0` is greater than `accruedFees0`.
  - ☐ Negative test
- `amount1` is greater than `accruedFees1`.
  - ☐ Negative test

Function call analysis

- `IBareVaultRegistry(vaultRegistry).getStrategyCreatorForVault(address(this))`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Strategist would be unable to collect fees if it reverted. Reentering is impossible given the `vaultRegistry` is a VaultRegistry contract, but it would have no impact in this context regardless.

Function: `tridentCLSwapCallback(uint256 amount0, uint256 amount1, byte[] None)`

This function is called by the TridentCL pool when performing a swap. It is used to pay Trident CL pool back tokens it is owed.

Inputs

- `amount0Owed`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `token0` balance of the vault.
  - **Impact**: The amount of `token0` to pay back to Trident CL.
- `amount1Owed`
  - **Control**: Full.

  - **Constraints**: Must be less than or equal to the `token1` balance of the vault.
  - **Impact**: The amount of `token1` to pay back to Trident CL.

Branches and code coverage (including function calls)

Intended branches

- Transfers tokens to the Trident CL pool. Indirectly tested when performing a swap.
  - ☑ Test coverage

Negative behavior

- `msg.sender` is not the pool.
  - ☐ Negative test

Function call analysis

- `token0.approve(address(pool), amount0Owed)`
  - **What is controllable?** `amount0Owed`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `BENTO.deposit( ... )`
  - **What is controllable?** `amount0Owed`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `token1.approve(address(pool), amount1Owed)`
  - **What is controllable?** `amount1Owed`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `BENTO.deposit( ... )`
  - **What is controllable?** `amount1Owed`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `tridentCLSwapCallback(int256 amount0Delta, int256 amount1Delta, byte[] None)`

This function is called by the TridentCL pool when performing a swap. It is used to pay Trident CL pool back tokens it is owed.

Inputs

- `amount0Delta`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `token0` balance of the vault.
  - **Impact**: The amount of `token0` to pay back to Trident CL.
- `amount1Delta`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the `token1` balance of the vault.
  - **Impact**: The amount of `token1` to pay back to Trident CL.

Branches and code coverage (including function calls)

Intended branches

- Transfers tokens to the Trident CL pool. Indirectly tested when performing a swap.
  - ☑ Test coverage

Negative behavior

- `msg.sender` is not the pool.
  - ☐ Negative test

Function call analysis

- `token0.approve(address(pool), uint256(amount0Delta))`
  - **What is controllable?** `amount0Delta`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `BENTO.deposit( ... )`
  - **What is controllable?** `amount0Delta`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `token1.approve(address(pool), uint256(amount1Delta))`

    - **What is controllable?** `amount1Delta`.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `BENTO.deposit( ... )`
    - **What is controllable?** `amount1Delta`.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `unpause()`

Calls the OpenZepplin PausableUpgradeable contract's `_unpause` function to flag the contract as unpaused. This function is intended to be called by the steer role only.

Branches and code coverage (including function calls)

Intended branches

- Flags the contract as unpaused.
    - ☐ Test coverage

Negative behavior

- Contract is already unpaused (this is enforced by `_unpause`'s `whenPaused` modifier).
    - ☐ Negative test
- Caller does not have `STEER_ROLE` role.
    - ☐ Negative test

Function call analysis

- `_unpause()`
    - **What is controllable?** None.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The contract will not be flagged as unpaused. This will not happen unless the negative behavior is intended, per the PausableUpgradeable contract.

Function: `withdraw(uint256 shares, uint256 amount0Min, uint256 amount1Min, address to)`

Withdraws the proportion of tokens in the vault and in the pool that corresponds to the shares burned.

Inputs

- `shares`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the number of shares owned by the sender. Otherwise, `_burn` will revert.
  - **Impact**: The amount of tokens withdrawn will be proportional to the number of `shares` burned.
- `amount0Min`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: If the amount of `token0` withdrawn is less than `amount0Min`, the function will revert.
- `amount1Min`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: If the amount of `token1` withdrawn is less than `amount1Min`, the function will revert.
- `to`
  - **Control**: Full
  - **Constraints**: None.
  - **Impact**: The withdrawn tokens will be sent to `to`.

Branches and code coverage (including function calls)

Intended branches

- Withdraws tokens and burns shares.
  - [x] Test coverage

Negative behavior

- Reverts if `shares` is zero.
  - [ ] Negative test
- Reverts if `shares` is greater than the number of shares owned by the sender.
  - [ ] Negative test

- Reverts if the amount of `token0` withdrawn is less than `amount0Min`.
  - ☐ Negative test
- Reverts if the amount of `token1` withdrawn is less than `amount1Min`.
  - ☐ Negative test

Function call analysis

- `totalSupply()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions.
- `_burn(msg.sender, shares)`
  - **What is controllable?** `msg.sender` and `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions.
- `_burnAndCollect(shares, _totalSupply)`
  - **What is controllable?** `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions.
- `token0.safeTransfer(to, amount0)`
  - **What is controllable?** `to`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions because ERC-20 tokens are being used.
- `token1.safeTransfer(to, amount1)`
  - **What is controllable?** `to`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the withdrawal will cancel. Reentrancy has no impact here and is not possible in normal conditions because ERC-20 tokens are being used.

5.6 Module: SushiMultiPositionLiquidityManager.sol

Function: `poke()`

Updates fees owed to each vault position. Given too many vault positions, this function may run out of gas.

Branches and code coverage (including function calls)

Intended branches

- Updates fees owed to each vault position.
  - ☑ Test coverage

Function call analysis

- `pool.burn(_lowerTick, _upperTick, 0)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Fees will not be updated. Function cannot reenter because `pool` is the trident liquidity pool contract.

Function: `tend(uint256 totalWeight, LiquidityPositions newPositions, byte[] timeSensitiveData)`

Adjusts the vault's positions to match the new positions.

Inputs

- `totalWeight`
  - **Control**: Full.
  - **Constraints**: Must be low enough that multiplication does not result in overflow.
  - **Impact**: The share of liquidity we want deposited, multiplied by 10,000.
- `newPositions`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The new positions to be set.
- `timeSensitiveData`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Encoded information of the `swapAmount` and `sqrtPriceLimitX96` used to mitigate MEV attacks.

Branches and code coverage (including function calls)

Intended branches

- Updates the position.
  - ☑ Test coverage
- Performs a swap.
  - ☑ Test coverage
- Creates a new position.
  - ☑ Test coverage

Negative behavior

- Reverts if `msg.sender` does not have the `MANAGER_ROLE` role.
  - ☐ Negative test
- Reverts if the contract is paused.
  - ☐ Negative test
- Reverts if TWAP-derived tick is too far from the current tick.
  - ☑ Negative test

Function call analysis

- `pool.slot0()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; compared against TWAP.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_checkVolatility(currentTick)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_burnAndCollect(1, 1)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_updatePositions(newPositions)`
  - **What is controllable?** `newPositions`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

If it reverts, the `tend` will revert. Reentering has no effect.

- `_swap( … )`
  - **What is controllable?** `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `pool.slot0()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_setBins( … )`
  - **What is controllable?** `FullMath.mulDiv(balance0, totalWeight, 1e4)`, `FullMath.mulDiv(balance1, totalWeight, 1e4)`, and `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.

5.7 Module: SushiSinglePositionLiquidityManager.sol

Function: `poke()`

Updates fees owed to the vault position.

Branches and code coverage (including function calls)

Intended branches

- Updates fees owed.
  - ☑ Test coverage

Function call analysis

- `pool.burn(_lowerTick, _upperTick, 0)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Fees will not be updated. Function cannot reenter because `pool` is the trident liquidity pool contract.

Function: `tend(uint256 totalWeight, LiquidityPositions newPositions, byte[] timeSensitiveData)`

Adjusts the vault's positions to match the new positions.

Inputs

- `totalWeight`
  - **Control**: Full.
  - **Constraints**: Must be low enough that multiplication does not result in overflow.
  - **Impact**: The share of liquidity we want deposited, multiplied by 10,000.
- `newPositions`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The new positions to be set.
- `timeSensitiveData`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Encoded information of the `swapAmount` and `sqrtPriceLimitX96` used to mitigate MEV attacks.

Branches and code coverage (including function calls)

Intended branches

- Updates the position.
  ☑ Test coverage
- Performs a swap.
  ☑ Test coverage
- Creates a new position.
  ☑ Test coverage

Negative behavior

- Reverts if `msg.sender` does not have the `MANAGER_ROLE` role.
  ☐ Negative test
- Reverts if the contract is paused.
  ☐ Negative test
- Reverts if TWAP-derived tick is too far from the current tick.
  ☑ Negative test

Function call analysis

- `pool.slot0()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; compared against TWAP.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_checkVolatility(currentTick)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_burnAndCollect(1, 1)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_updatePositions(newPositions)`
  - **What is controllable?** `newPositions`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_swap( … )`
  - **What is controllable?** `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `pool.slot0()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.
- `_setBins( … )`
  - **What is controllable?** `FullMath.mulDiv(balance0, totalWeight, 1e4)`, `FullMath.mulDiv(balance1, totalWeight, 1e4)`, and `swapAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the `tend` will revert. Reentering has no effect.

5.8 Module: VaultRegistry.sol

Function: `createVault(byte[] _params, uint256 _tokenId, string _beaconName, address _vaultManager, string _payloadIpfs)`

Creates a new vault, taking the given strategy and registering it with the registry. Returns the address of the newly created vault. This function is intended to be called by any user.

Inputs

- `_params`
  - **Control**: Full.
  - **Constraints**: Must be the correct size for the vault parameters.
  - **Impact**: The parameters for the vault being created. If this value is not the correct size, decoding it will abort in the vault constructor.
- `_tokenId`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The ID of the token to be added to the tokens list of the given address. Because this value has no constraints, it does not need to already exist or be consecutive.
- `_beaconName`
  - **Control**: Full.
  - **Constraints**: Must exist as a key of `beaconAddresses` or the function will revert with "Beacon is not present".
  - **Impact**: The address of the beacon used to create the vault.
- `_vaultManager`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The address of the vault manager that will manage the vault being created.
- `_payloadIpfs`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: A setting for the vault being created. This value is not used by the registry but is stored in the `VaultData`.

Branches and code coverage (including function calls)

Intended branches

- Creates a vault and registering it.
  - ☑ Test coverage

Negative behavior

- Beacon is not present.
  - ☑ Negative test
- Vault `_params` is invalid.
  - ☐ Negative test

Function call analysis

- `new BeaconProxy( … )` and `newVault.initialize(_vaultManager, owner(), internalGovernance, _params)`
  - **What is controllable?** `_vaultManager` and `_params`.
  - **If return value controllable, how is it used and how can it go wrong?** It is not controllable but is the address of the contract that is deployed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the vault will not be created. If it reenters, it would be equivalent to calling `createVault` twice (i.e., no impact).
- `_addLinkedVaultsEnumeration(_tokenId, address(newVault), _payloadIpfs, _beaconName)`
  - **What is controllable?** `_tokenId`, `_payloadIpfs`, and `_beaconName`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** It cannot reasonably reenter or revert (e.g., it is impossibly impractical for the `tokenId` addition to overflow).

Function: `pause()`

Calls the OpenZepplin PausableUpgradeable contract's `_pause` function to flag the contract as paused. This function is intended to be called by the pauser role only.

Branches and code coverage (including function calls)

Intended branches

- Flags the contract as paused.
  - ☑ Test coverage

Negative behavior

- Contract is already paused (this is enforced by `_pause`'s `whenNotPaused` modifier).
  - ☐ Negative test

- Caller does not have `PAUSER_ROLE` role.
  ☑ Negative test

Function call analysis

- `_pause()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The contract will not be flagged as paused. This will not happen unless the negative behavior is intended, per the PausableUpgradeable contract.

Function: `unpause()`

Calls the OpenZepplin PausableUpgradeable contract's `_unpause` function to flag the contract as unpaused. This function is intended to be called by the pauser role only.

Branches and code coverage (including function calls)

Intended branches

- Flags the contract as unpaused.
  ☑ Test coverage

Negative behavior

- Contract is already unpaused (this is enforced by `_unpause`'s `whenPaused` modifier).
  ☐ Negative test
- Caller does not have `PAUSER_ROLE` role.
  ☑ Negative test

Function call analysis

- `_unpause()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The contract will not be flagged as unpaused. This will not happen unless the negative behavior is intended, per the PausableUpgradeable contract.

Function: `updateVaultState(address _vault, VaultState _newState)`

Updates the vault state and emits a VaultStateChanged event. This function is intended to be called by the registry owner.

Inputs

- `_vault`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The address of the vault to be updated.
- `_newState`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The new state of the vault.

Branches and code coverage (including function calls)

Intended branches

- Updates the vault state.
  - [x] Test coverage

Negative behavior

- Caller is not the owner.
  - [ ] Negative test

6 Audit Results

At the time of our audit, the audited contracts were not deployed to mainnet EVM.

During our audit, we discovered seven findings. Of these, one was low risk and the rest were suggestions (informational). Steer 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 audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code 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.

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.